2-9-1993

Minimization of Permuted Reed-Muller Trees and Reed-Muller Trees for Cellular Logic Programmable Gate Arrays

Lifei Wu
Portland State University

AN ABSTRACT OF THE THESIS OF Lifei Wu for the Master of Science in Electrical and Computer Engineering presented February 9, 1993.

Title: Minimization of Permuted Reed-Muller Trees and Reed-Muller Trees for Cellular Logic Programmable Gate Arrays

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

[redacted]

Marek A. Perkowski, Chair

[redacted]

Małgorzata E. Chrzanowska-Jeske

[redacted]

Jing-ke Li

The new family of Field Programmable Gate Arrays, CLI 6000 from Concurrent Logic Inc realizes truly Cellular Logic. It has been mainly designed for the realization of data path architectures. However, the realizable logic functions provided by its macro-cells and their limited connectivity call also for new general-purpose logic synthesis methods. The basic cell of CLi 6000 can be programmed to realize a two-input multiplexer (A*B + C*$\bar{B}$), an AND/EXOR cell (A*B $\oplus$ C), or the basic 2-input AND, OR and EXOR gate. This suggests to using these cells for tree-like expansions. These "cellular logic" devices require regular connection patterns in the netlists resulting from logic

synthesis. This thesis presents a synthesis tree searching program PROMPT, which generates AND/EXOR tree circuits from given Boolean functions. Such circuits have the property that the gate structures are AND/EXOR (A*B ⊕ C), AND and EXOR which could be realized by the CLI6000 cells. Also, the connection way in the circuit is that usually the output of one level gate is the input of the next level gate of the tree. This matches ideally to the architecture of the CLI6000 bussing network where the macrocells have only connections to their neighboring cells. PROMPT is based on the Davio expansions (an equivalent of the Shannon expansions for the EXOR gates) as its Boolean decomposition methods. The program includes three versions: *exact version, heuristic version and fixed-variable version.*

The *exact version* of PROMPT generates the Permuted Reed-Muller Tree circuit which has the minimum number of gates. Such tree circuit is obtained by searching through all possible combinations of the expansion variable orders to get the one which needs the least number of gates.

The *heuristic version* of PROMPT is designed to decrease the time complexity of the search algorithm when dealing with logic functions having many input variables. It generates a Permuted Reed-Muller Tree which may not have the minimum number of gates. However, the tree searching time in this version decreases tremendously compared to the time necessary in the exact version.

The *fix-variable version* is developed to generate Reed-Muller Tree circuits. Such circuits will have the same expansion variables at the same tree level, so they can be easier routed after the placement to the CLI6000 chips.

In short, the program PROMPT generates the PRM and RM tree circuits which are particularly well matched to both the realization of logic cell and connection structure of the CLI6000 device. Thus, the PRM and RM circuits can be easily placed and routed on the CLI6000 FPGAs.

MINIMIZATION OF PERMUTED REED-MULLER TREES AND REED-MULLER TREES FOR CELLULAR LOGIC PROGRAMMABLE GATE ARRAYS

by

LIFEI WU

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

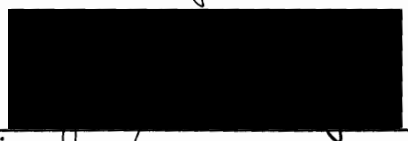
Portland State University
1993

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Lifei Wu presented February 9, 1993.

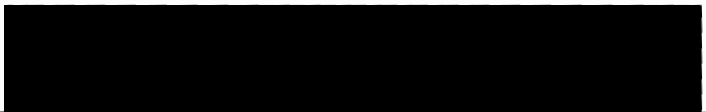
Marek A. Perkowski, Chair

Malgorzata Chrzanowska-Jeske



Jing-ke Li

APPROVED:



Rolf Schaumann, Chair, Department of Electrical Engineering

Roy W. Koch, Vice Provost for Graduate Studies and Research

ACKNOWLEDGEMENTS

I would like to thank my parents and my grandparents for their love and spiritual support.

I would like to thank Marek A. Perkowski, Chair of the Committee, for his initial inspiration and professional guidance throughout the research.

I would also like to thank other Committee members: Malgorzata Chrzanowska-Jeske and Jing-ke Li for their assistance and numerous suggestions in the preparation of the thesis.

I am grateful to Shirely Clark and Laura Riddell for their provision of all kinds of help during my research period.

I am grateful to Ingo Schäfer for helping me to start the program and giving me suggestsions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In the last decade there has been a tremendous effort to develop design automation tools for the development of increasingly complex digital circuits. To decrease additionally the necessary design time, synthesis tools for Application Specific IC's (ASICs) based on high-level description languages (HDL, such as VHDL and Verilog) have been introduced.

The upcoming of Field Programmable Gate Arrays (FPGAs) allowed to further decrease the development and testing phase for new products. While for ASICs the process of making and testing of a single development iteration of a new circuit takes several monthes due to the necessary chip fabrication, FPGAs allow for an immediate realization of the logic function with following testing.

The logic synthesis methods developed for FPGAs have been based on algebraic decomposition methods [123456]. However, it is known that logic synthesis methods based on Boolean decomposition methods can produce better results[7]. Moreover, like the logic synthesis tools for ASICs, the FPGA logic synthesis concepts have been based on the "unate paradigm" [7]. The "unate paradigm" implies, that most of the logic functions occurring in real life design have a minimal circuit realization based on AND and OR gates. However, arithmetic functions like adders, multipliers, counters, signal processing functions, and error correcting logic have a smaller circuit realization if the EXOR gate is incorporated. The synthesis incorporating the EXOR gate have been neglected because the EXOR gate was perceived to be slower and having a larger circuit area.

The upcoming of FPGAs like the Xilinx Table-Look-Up (TLU) architecture, the Actel multiplexer based (MB) architecture and recently the fine grain FPGAs from Concurrent Logic [8], Algotronix [9] and Texas Instruments[10] allow the implementation of the EXOR gate without any speed or circuit size penalty in comparison to the AND and OR gates.

This thesis introduces a decomposition algorithm for the CLi 6000 FPGA series from Concurrent Logic [8] to overcome the disadvantages of current synthesis tools with respect to the synthesis for EXOR gates in FPGAs. Moreover, the presented synthesis concept generates a regular circuit structure which can be easily placed and routed on the regular architectures provided by the new fine grain (cellular, sea of gates) type FPGAs.

The basic cell of CLi 6000 can be programmed to realize a two-input multiplexer (A*B + C*$\overline{B}$), an AND/EXOR cell (A*B $\oplus$ C), or the basic 2-input AND and OR gate. The multiplexer and the AND/EXOR cell realization are the two most efficient realizations of combinational functions with a Cli 6000 macrocell. This suggests to develop synthesis methods based on the AND/EXOR gate structure.

The realization of a logic function, using AND and EXOR gates, based on Reed-Muller canonic expansions has the characteristic of being easily testable [11]. Moreover, the regular structure of a Reed-Muller Tree circuit is that the basic gate structures are AND/EXOR (A*B $\oplus$ C), AND and EXOR, and connections between the gate structures are only from one level of the tree to another level. Therefore, such a circuit realization is ideally suitable for the implementation on FPGAs having limited wiring resources. Such a circuit realization is in particular beneficial for the CLi 6000 FPGA series from Concurrent Logic having a cellular architecture, where each macrocell has connections only to its four neighbours (plus limited local busses).

PROMPT is a synthesis tree searching algorithm presented in this thesis which generates Reed-Muller Tree from a given Boolean function. PROMPT includes three

versions: *exact version, heuristic version and fixed-var version.*

The *exact version* of PROMPT generates the minimal Permuted Reed-Muller Tree[12] for a given Boolean function. The Permuted Reed-Muller Trees included all possible expansion order combinations for the variables. It means that different expansion variables could appear at the same tree level. The goal of the *exact version* is to find the minimum gate number Permuted Reed-Muller circuit for the given Boolean function. However, it is traded off with the non-regular structure of variables connecting networks in the different subtrees. Thus, the following placement and routing problem is of higher complexity.

The limited bussing network of the CLi 6000 requires some additional restrictions to the presented exact tree searching algorithm. To accommodate this restrictions to allow an easy placement and routing of the macrocells the *fixed-var version* of PROMPT is introduced. In the fixed-var searching algorithm the data-select variables in one particular level of the tree is there restricted to be the same in each node. In other words, the fix-var version will generate Reed-Muller Tree.

The *heuristic version* of PROMPT is designed to decrease the time complexity of the search algorithm when dealing with logic functions having many input variables. It generates a Permuted Reed-Muller Tree which may not have the minimum number of gates. However, the tree searching time in this version decreases tremendously compared to the time necessary in the exact version.

The following gives an outline of the organization of this thesis. First, in Chapter II, the basic architecture of the CLi 6000 series is reviewed and the main restrictions that lead to the presented synthesis methods are pointed out. Chapter III gives the theoretical background to the developed concept. There, the Davio and Shannon Expansions and their relation to the synthesis for the CLi 6000 are presented.

Chapter IV introduces PROMPT, the logic synthesis algorithm generating an AND-EXOR circuit based on the application of expansion given in Chapter III. An exact algorithm and a heuristic method to decrease the computation time are presented. Chapter V outlines the technology mapping of the PRM circuit and RM circuit to the Cli 6000 chips. Chapter VI provides the testing data and analyses the results obtained from the implementation of PROMPT. Finally, Chapter VII presents the conclusions derived from the presented logic synthesis concept.

CHAPTER II

ARCHITECTURE OF CLI6000 SERIES

II.1 GENERAL DESCRIPTION

The introduction of Field Programmable Gate Arrays (FPGAs) in the last decade had a very important impact on the development of new circuit concepts and logic synthesis methods. FPGAs allow for an immediate realization and testing of the circuit under design without going through the process of a complete chip production necessary for custom or PGA designs. Thus, FPGAs allowed to decrease the development and testing phase for products. However, until recently FPGAs did not offer a similar density and system clock rate as Application Specific ICs (ASICs). Therefore, they were limited to only few applications.

The Cli6000 Series is a new generation of Field Programmable Gate Arrays introduced by Concurrent Logic Inc, which overcomes the above stated drawbacks. Its general architecture is based on an array of logic cells. In contrast to other FPGAs, like the Actel's MB based or Xilinx's TLU based approaches, the logic cells in the CLi 6000 series can realize functions of only up to three input variables. Therefore, the architecture is also called fine grain cellular array FPGA. Similarly, the Algotronix CAL 1024 and the Texas Instruments TPC-10 employ such fine grain cellular architectures. Because this thesis introduces synthesis methods especially suited for the CLi 6000 series, this Chapter reviews the basic features of this architecture.

The CLi6000 Series employs a patented, symmetrical architecture consisting of many small yet powerful logic cells connected to a flexible bussing network and sur-

rounded by programmable I/O. The Concurrent Logic architecture was developed to provide the highest levels of performance, functional density and design flexibility in a FPGA. The cell's small size allows for the realization of arrays with a large number of cells. For example, the Cli6010 has 6400 logic cells while the largest Xilinx400 chip has only 900 cells, so that the lower cell complexity is traded off for the larger number of cells. A simple, high-speed bussing network offers fast, efficient communication over medium and long distances. Thus, the CLi 6000 series provides the density and performance of custom gate arrays without the prototyping and debugging delays necessary for mask-programmed devices.

II.2 THE BUSSING NETWORK

Figure 1 shows one CLi cell. Figure 2 illustrates the 8x8 cells with busses.

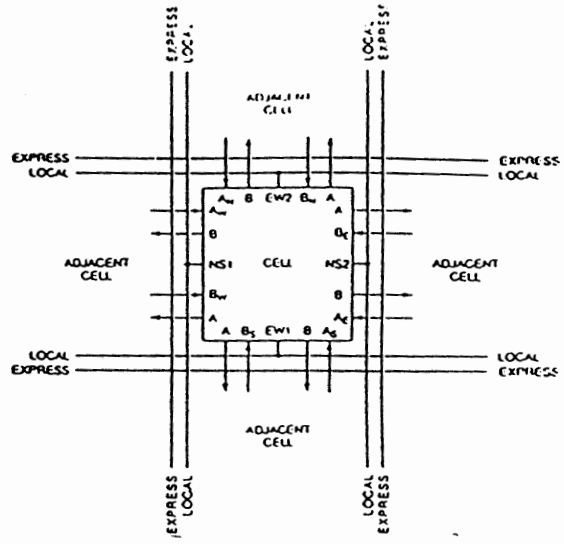


Figure 1. Cell-to-cell and cell-to-bus connections.

To perform the logic functions, each cell takes three inputs: two inputs from any of

the four neighboring cells (A_E or A_S or A_W or A_N and B_E or B_S or B_W or B_N), one input from a bus (EW_1 or EW_2 or NS_1 or NS_2) and could provide two outputs A and B to all neighbors, one output to bus (EW_1 or EW_2 or NS_1 or NS_2).

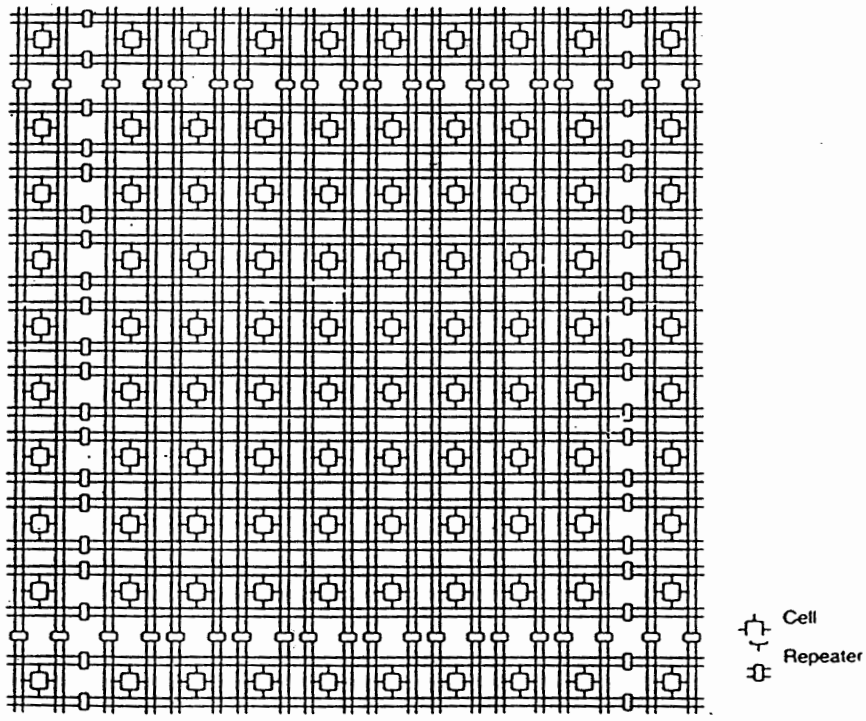


Figure 2. Bussing Network.

As one can observe from Figure 2, the CLi 6000 series has a very limited wiring resources. Therefore, a tree-like circuit that has connections only between two levels of the circuit matches ideally such a structure. Moreover, a tree circuit having at a particular level the same input variable to each module can efficiently take advantage of the bus network. In Chapter IV an algorithm to generate such circuits will be presented.

II.3 THE CLI6000 MACROCELL

Due to the small size of the macrocells on the CLi 6000 FPGA, only a limited set of logic functions having up to three inputs can be realized with a single cell. Figure 3. gives the 20 configurations for combinational logic functions that are possible with one macrocell.

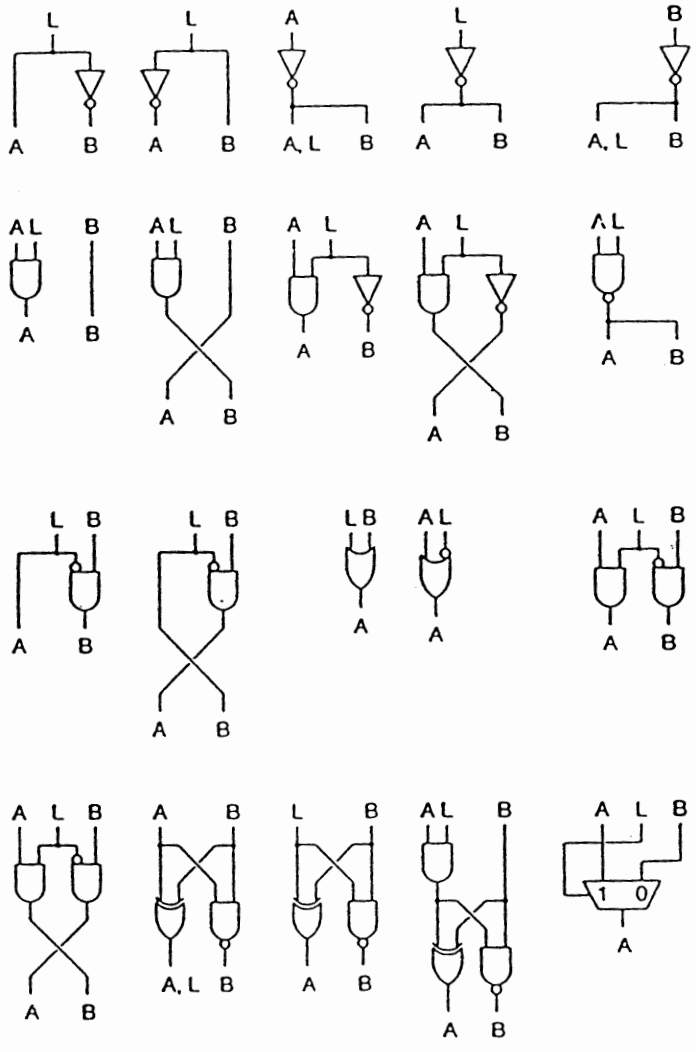


Figure 3. Combinatorial States of CLI Macrocell.

As one can observe from Figure 3, there are four basic logic functions AND, OR, EXOR and AND-EXOR. Additionally, for configurations having three inputs, one input has to be from the local bus. Thus, synthesis methods that take these restrictions into account have to be developed. This thesis investigates logic synthesis methods that efficiently minimize the number of macrocells by taking advantage of the powerful AND-EXOR cell realization. To take the connectivity of the cells into account we propose a synthesis method that generates a tree circuit. Such a circuit has the property that the gate structures are AND/EXOR (A*B $\oplus$ C), AND and EXOR which could be realized by the CLI6000 cells. Also, the connection way in the circuit is that the output of one level gate is the input of the next level gate. This matches ideally to the architecture where the macrocells have only connections to their neighboring cells. Moreover, in Chapter IV we employ a further adaptation of the presented synthesis algorithm to match the availability of the local busses. Using this algorithm, the generated AND/EXOR circuit has the same input variable in a particular level. Therefore, the realization of the circuit is easily routable.

The next Chapter introduces an efficient Boolean decomposition method that allows for the decomposition of a logic function to the functions that are realizable with a single macrocell.

CHAPTER III

THE DAVIO EXPANSIONS AND THEIR CIRCUIT REALIZATIONS

The general objective of decomposition methods in logic synthesis is to decompose a given set of functions into smaller subfunctions that can be realized by certain gate structures such that the final circuit realization is optimized for speed and area. Usually, a large logic function is difficult to analyze and to find a small circuit realization for it. One way to solve the problem is to decompose the initial logic function into smaller blocks which are easier to implement. There are two basic approaches to decomposition [7]: the algebraic factorization and the Boolean decomposition. The algebraic decomposition methods are based on the factoring and extraction of common subfunctions. Boolean decomposition methods take advantage of the structure of the function to be decomposed. Because they operate on the whole functions they are computationally more expensive than the algebraic methods. However, it was observed [7] that they lead to better results than the algebraic methods. Therefore, the multilevel synthesis tools like RENO and MISII make use of algebraic methods to find a local minimum and try then to apply Boolean decomposition methods to find a lower local minimum.

One of the most fundamental concepts for the decomposition of a logic functions is the Shannon expansion. The Shannon expansion can always be applied to a logic function in contrast to other types of Boolean decompositions like the Ashenhurst [13] or the Curtis [14] decomposition. These decompositions can be only applied to logic functions belonging to a certain class, like the class of disjoint decomposable functions.

Therefore, this Chapter reviews the concepts of the Davio expansions over the Galois Field (2) (GF2) and shows its circuit realizations. It will be shown, that the Davio

expansion suits ideally for the decomposition of logic functions to subfunctions that can be ideally realized with the CLi 6000 FPGA series.

III.1 DAVIO EXPANSIONS

The well-known Davio expansions is given by [1516]

$$f(x_1,...x_i,...x_n) = x_i \cdot f(x_1,..,x_i=1,\ ,...x_n) \oplus \overline{x_i} \cdot f(x_1,..,x_i=0,...x_n) \quad (1)$$

By applying the rules $\overline{a} = 1 \oplus a$ and $a = 1 \oplus \overline{a}$ one obtains the two Davio expansions [15]:

$$f(x_1,...x_i,...x_n) = f(x_1,...x_i=0,...x_n) \oplus x_i \cdot [f(x_1,...x_i=0,...x_n) \oplus f(x_1,...x_i=1,...x_n)] \quad (2)$$

and

$$f(x_1,...x_i,...x_n) = f(x_1,...x_i=1,...x_n) \oplus \overline{x_i} \cdot [f(x_1,...x_i=0,...x_n) \oplus f(x_1,...x_i=1,...x_n)] \quad (3)$$

in short form:

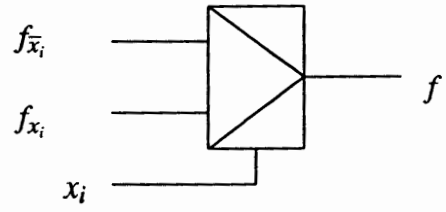
$$f = x_i f_{x_i} \oplus \overline{x}_i f_{\overline{x}_i} \quad (4)$$

$$f = f_{\overline{x}_i} \oplus x_i \left[f_{x_i} \oplus f_{\overline{x}_i} \right] = f_{\overline{x}_i} \oplus x_i\, g \quad (5)$$

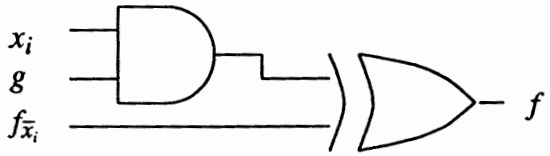
$$f = f_{x_i} \oplus \overline{x}_i \left[f_{x_i} \oplus f_{\overline{x}_i} \right] = f_{x_i} \oplus \overline{x}_i\, g \quad (6)$$

The circuit realization of Equation (4) is given by a multiplexer gate while Equations (5) and (6) describe an AND-EXOR gate structure, see Figure 4.

a. Circuit Realization of Equation (4):



b. Circuit Realization of Equation (5):



c. Circuit Realization of Equation (6):

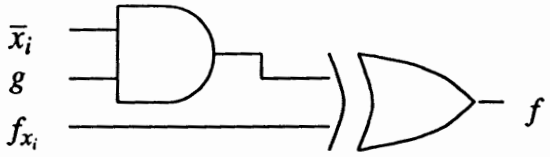


Figure 4. Circuit realizations of the three Shannon expansions.

It can be observed from Figure 4 that the circuit realization of the three expansions correspond to the realizable functions of a macrocell of the Concurrent Logic CLi 6000 FPGA series. Therefore, the Davio expansions are ideally suited for the decomposition of Boolean functions with respect to the realization with the CLi 6000 series.

One can observe, that be applying the three expansions in different orders one obtains different tree forms. By "flattening" of the tree forms, which will be described in the next section, one obtains the Reed-Muller forms [151718].

The next section investigates the different forms that can be obtained by applying the Davio expansions.

III.2 CIRCUIT REALIZATIONS OBTAINED BY THE DAVIO EXPANSIONS

III.2.1 Multi-Level Circuit Realizations

Definition III.1: The literal of a variable x_i can be either positive (x_i) or negative ($\overline{x_i}$) form.

Definition III.2: The polarity of a variable is "1" for a positive literal and "0" for a negative literal.

It follows from Equation (4)-(6) and Definition III.2, that different tree forms are obtained when the Davio expansions are applied in different orders. There are four possible ways of applying the three Davio expansions [151718]. The following Definitions give the various tree forms obtained by applying the Davio expansions.

Definition III.3: Reed-Muller Tree (RMT) -- The expansion tree in which all variables appear positive polarity only and in each subtree the decomposing variable has the same order (See Figure 5).

During the decomposition, if only rule (5) is used repeatedly for some fixed order of expansion variables, the RM trees are created.

Definition III.4: Permuted Reed-Muller Tree (PRMT) -- The expansion tree in which all variables appear in positive polarity but in each subtree the decomposing variables could be in a different order (See Figure 6).

During the decomposition, if only rule (5) is used repeatedly, but the order of expansion variables is not fixed, the PRM trees are created.

Definition III.5: Generalized Reed-Muller Tree (GRMT) -- The expansion tree in which each variable appears only either in positive or negative polarity (See Figure 7).

During the decomposition, if for every variable one uses either rule (5) or rule (6) , the GRM trees are created.

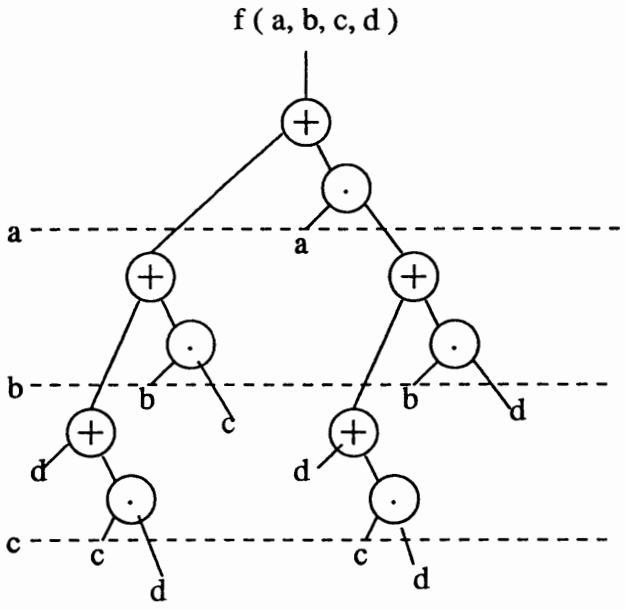


Figure 5. RMT.

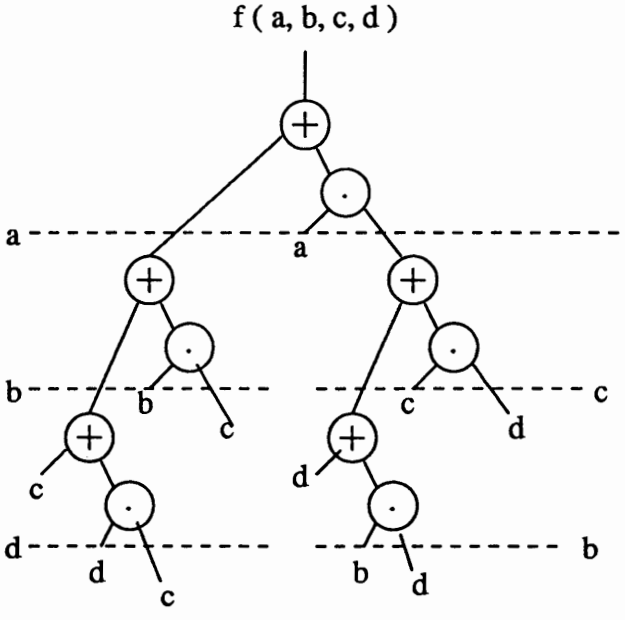


Figure 6. PRMT.

Definition III.6: Kronecker Reed-Muller Tree (KRMT) -- The expansion tree in which all variables appear in either positive or negative, or both polarities, but having a single fixed order of expansion variables in the tree levels (See Figure 8).

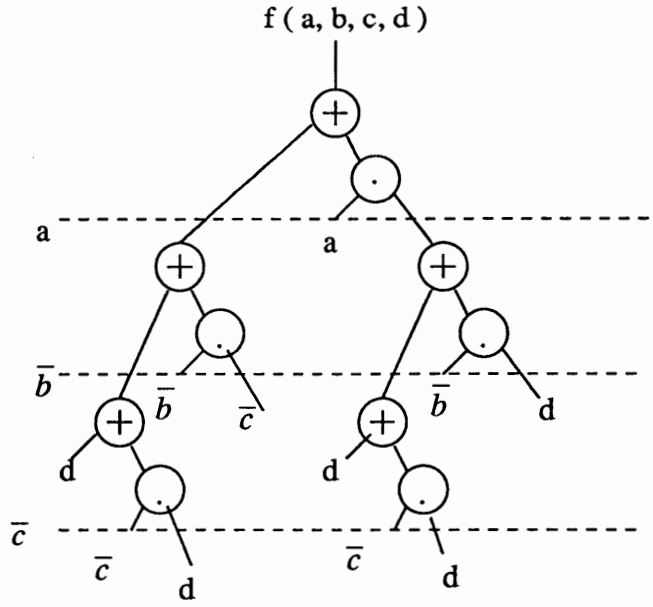


Figure 7. GRMT.

During the decomposition, if for every variable one uses either rule (4), rule (5), or rule (6) without regard to consistency, the KRM trees are created [16].

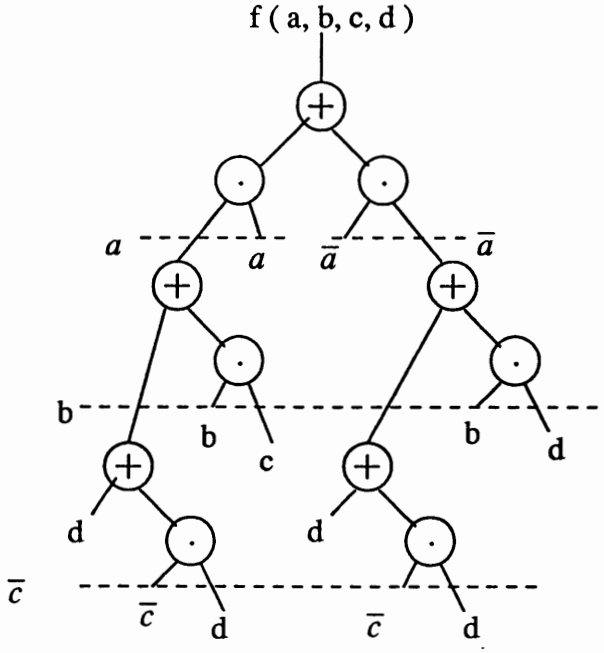


Figure 8. KRMT.

Definition III.7: Pseudo-Kronecker Reed-Muller Tree (PKRMT) -- The expansion tree in which all variables appear in either positive or negative, or both polarities and having all possible orders of expansion variables (See Figure 9).

During the decomposition, if rule (4), (5) and (6) are used , but in each subtree there is a choice of a rule, the PKRM trees are generated [16].

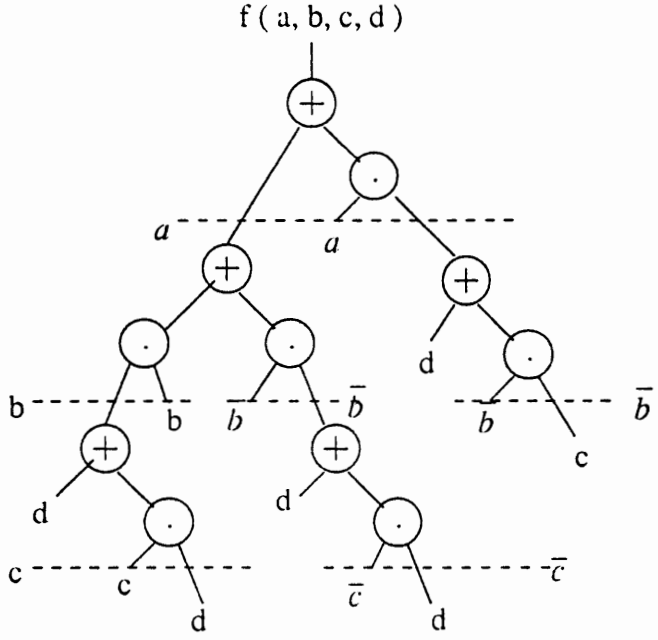


Figure 9. PKRMT.

For allowing various orders of variables in subtrees or using different rules in subtrees, an even wider family of trees could be created.

III.2.2 Two Level Circuit Realizations

The expansion formulas applied to various variables and the resulting subfunctions f_j generate different multi-level tree circuits as described in the previous section.

The obtained tree circuits can be flattened to a two level form which can be realized by an AND-EXOR circuit.

Definition III.8: Flattening -- Applying rule $a\ (\ b \oplus c\ ) = a\ b \oplus ac$ recursively to a multilevel network until a 2-level network (ESOP) is created.

Definition III.9: Reed-Muller (RM) form is an ESOP obtained by flattening of a RM Tree.

For example, assuming a four variables input function, the RM form expression is of the following form:

$$F_1\ (a,\ b,\ c,\ d\ ) = a \oplus ab \oplus abc \oplus cd \oplus abcd$$

Definition III.10: Generalized Reed-Muller (GRM) form is an ESOP obtained by flattening of a GRM Tree.

For example, assuming a four variables input function, the GRM form expression is of the following form:

$$F_{2(}\ a,\ b,\ c,\ d\ ) = a \oplus a\bar{b} \oplus a\bar{b}c \oplus c\bar{d} \oplus a\bar{b}c\bar{d}$$

Definition III.11: Kronecker Reed-Muller (KRM) form is an ESOP obtained by flattening of a KRM Tree.

Definition III.12: Pseudo-Kronecker Reed-Muller (PKRM) form is an ESOP obtained by flattening of a PKRM Tree.

The tree structure obtained by the RM tree is ideally suited for the mapping to the CLi 6000 FPGA series. However, the restricted connectivity between neighboring cells and the limited bus resources require a modification of the general RM tree circuit. Therefore, in the next Chapter the general tree circuit algorithm will be presented and its modification to allow the generation of circuits that are additionally easily routable.

CHAPTER IV

DESCRIPTION OF PROMPT

PROMPT is the synthesis algorithm for the calculation of the Reed-Muller Tree for a given Boolean function having the minimal number of AND/EXOR gates. As mentioned, PROMPT consists of three options. The *exact* option determines the absolute minimal Permuted Reed-Muller Tree for the underlying Boolean function. In order to reduce the complexity of the implied exhaustive search, the *heuristic* option has been also developed that leads to a quasi minimal Permuted Reed-Muller Tree. Finally, a third option of PROMPT has been developed to provide Reed-Muller Tree that is especially suited for the technology mapping to the CLi 6000 series of Concurrent Logic.

In the next sections the three different synthesis concepts are illustrated.

IV.1 EXACT PRM TREE SEARCHING ALGORITHM

IV.1.1 Basic Idea

The *Reed-Muller (RM) expression* is an exclusive sum of products of positive (non-complemented) input variables. Therefore, only equation (5) should be used during the decomposition.

A given Boolean function can be decomposed using Equation (5):

$$f = f_{\bar{x}_i} \oplus x_i \cdot g_{x_i} \qquad \text{where } g_{x_i} = f_{\bar{x}_i} \oplus f_{x_i}$$

The two subfunctions $f_{\bar{x}_i}$ and g_{x_i} are independent of the variable x_i. The AND-gate takes (x_i) and g_{x_i} as its two inputs. The EXOR-gate takes $(f_{\bar{x}_i})$ and the output of the AND-gate as its two inputs. Their relationship is shown in Figure 10. For each subfunc-

tion $f_{\overline{x_i}}$ and g_{x_i}, choosing another variable, such as (x_j), the Davio Expansion is used to continue the decomposition. After the further decomposition, both functions $f_{\overline{x_i}}$ and g_{x_i} will be decomposed into another two subfunctions, which are connected by AND-gate and EXOR-gate also, but at one level down in the tree (See Figure 11). The decomposition is repeated recursively until the input functions are all trivial (0, 1, x_j). After completing the decomposition, a tree circuit is created. The output of the tree circuit is the original function f. In the tree circuit, all input variables appear in positive polarity, they are connected by AND-gates and EXOR-gates.

In other words, after the decomposition, the Boolean function is represented by multi-level AND, EXOR circuit, where the inputs are intermeditate functions of positive literals. Let us observe that all gates have a fan-in of only two. This kind of decomposition is ideally suited to a regular array realization of the combinational logic, such as those offered in "cellular logic" or new Field Programmable Gate arrays like those from Concurrent Logic[8] and Algotronix[9].

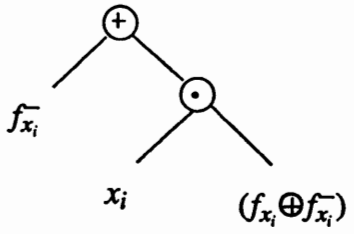


Figure 10. Function decomposition by Davio expansion.

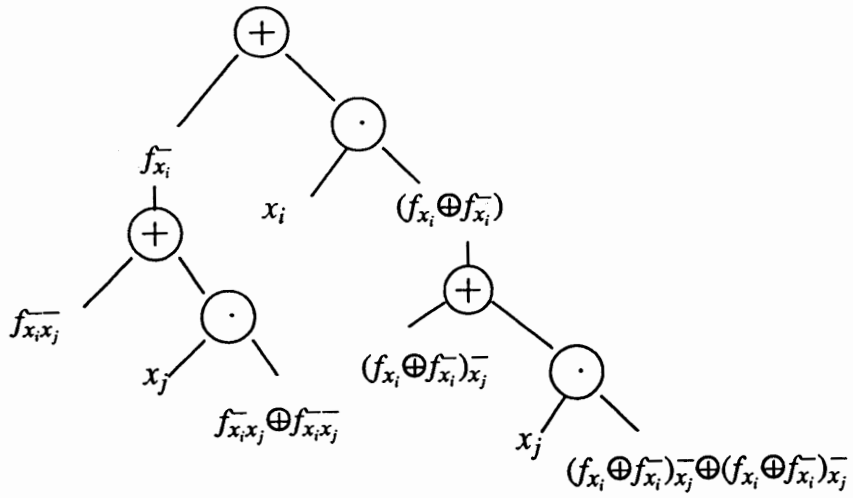


Figure 11. Expansion tree with two levels.

The diagram from Figure 12 illustrates all possible choices of decomposing a n input variable function using Davio Expansion. This diagram describes all applications of Davio Expansion with respect to all possible variables in all subtrees.

Here, we use Figure 12 as an example to explain the steps of how to apply the Davio Expansion to completely decompose a function.

Suppose the input function f has n variables.

(1) At the first level of the tree (the root of the tree), there is only one node, which describes the input Boolean function f.

(2) The second level is obtained by the decomposition of function f in variable x_i. Because there are n variables, so there are n possible choices for the decomposition at this level. Therefore, at the second level, there are 2n nodes. Each pair of nodes corresponds to the two subfunctions: $f_{x_i}^-$ and g_{x_i}, which are obtained by decomposing a variable x_i from f using Equation (5) at the root (first level).

(3) Each node at the second level could be further decomposed by using Davio Expansion form to decompose other variables as their parent-node did, but there are less

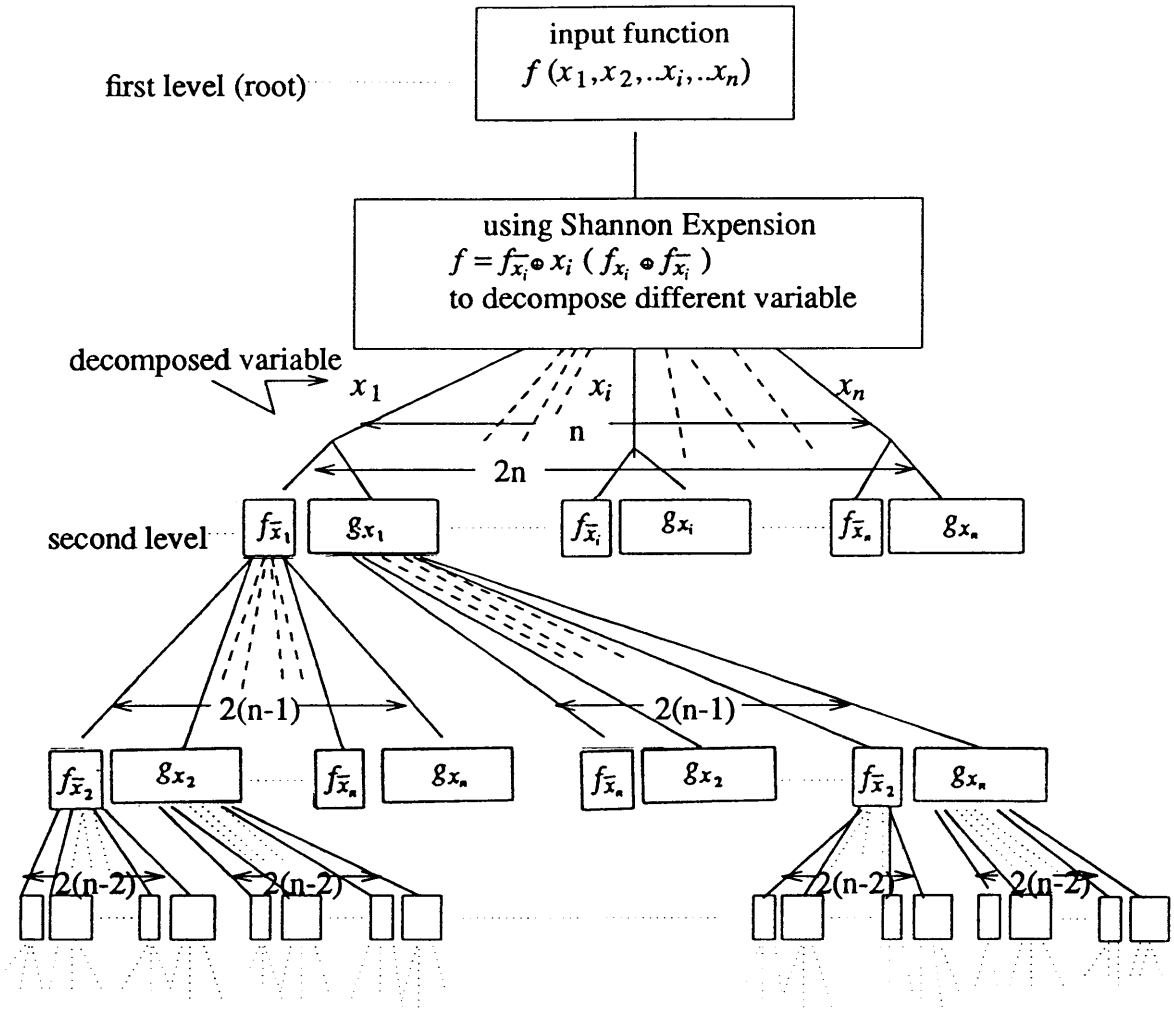


Figure 12. Diagram for the full_tree searching.

decomposing choices than in the parent-node (because the number of choices is determined by the number of remaining variables in the subfunction).

(4) It is clear that at the third level, there are 2n*(n-1) nodes (each node contains a subfunction: f_{x_n} *or* g_{x_n}).

(5) Each node continues the decomposing process until all variables are decomposed (no more variable remains in the subfunction).

(6) From calculation, at the last level, there are total $n!*2^n$ nodes. In the worst case, the tree has a maximum of n levels.

Even though all variable's decomposition is according to the same form

$$f = f_{\bar{x}_i} \oplus x_i \cdot g_{x_i}$$

the variable's decomposing order still makes the results (the total number of gates used to represent the original Boolean function) different. The goal of the exact PRM tree searching program is to find the best variable decomposing order and obtain the optimum solution.

IV.1.2 Algorithm for the PRM tree searching

In order to find the minimal PRM Tree, checking all possible decomposing choices of the function is necessary. A depth first algorithm has been adapted to travel through all possible variable selections during the decomposition.

The advantage of using a depth-first search is that it requires less memory, since only the nodes on the current path from the root to the leaf are stored. This contrasts with the breadth-first search, in which the complete tree has to be stored..

The depth-first algorithm that follows is a recursion function, *decompose(f)*. This function is used to realize the decomposition.

```
decompose ( functpt )
        /* functpt is a pointer which points to
                the input function ( cube array ) */
{
        var = select_variable ( functpt );
        f_x = subfuncl(functpt , var);
        g_x = subfuncr(functpt , var);

        if ( f_x not trivial )
                decompose( f_x );
        if ( g_x not trivial )
                decompose( g_x );
}
```

The function *decompose(functpt) takes a Boolean function as its input, in the form of cube array which is pointed by pointer functpt.* The function *select_variable(functpt)* picks a variable from the inputs. For this selected variable, using the Davio Expansion to decompose the function, the subfunctions f_x and g_x are obtained. Next it has to be identified if the subfunctions f_x and g_x are already trivial, otherwise another variable has to be selected to decompose the subfunctions. The decomposition will continue on all subfunctions until all inputs are trivial.

The routines in the function *decompose()*:

select_variable(functpt) -- Select a variable for the decomposition of the function.

subfuncl(functpt,var) -- compute the subfunction of function f which responds to $f_{\overline{x}_i}$. *Subfuncl(functpt,var)* returns a pointer which points to the cube array of $f_{\overline{x}_i}$.

subfuncr(functpt,var) -- compute the subfunction of function f which responds to $g_{x_i} = f_{\overline{x}_i} \oplus f_{x_i}$. *Subfuncr(functpt,var)* returns a pointer which points to the cube array of g_{x_i}.

f is trivial if $f \in \{ 0, 1, x, \overline{x} \}$.

To obtain the exact minimal solution, a complete tree search has been implemented.

Exact searching -- searching the whole tree to generate all possible combinations of different variable orders for the decomposition and finding the best result. For a given n input variable function, the exact search will involve a total of $n!*2^n$ possible paths.

Definition IV.1: Decomposing Tree -- The tree describing all possible decompositions.

The pseudo code for the exact tree search is given by the function *decomposeT(funcpt, variable). DecomposeT(funcpt, variable)* generates the decomposing tree.

```
decomposeT ( functpt, variable )
{
        int i;
        int undecomposed = 0; /* flag */
        int decomposed = 1;   /* flag */
                /* This flag is used to indicate variable
                   decomposition states. If the variable has
                   been decomposed, set the flag to 1, otherwise,
                   set flag to 0.    */

      f_x = subfuncl(functpt, variable);

        if ( f_x not trivial ){
                for ( i = 0; i < literals; i++ ){
                        if ( var[i] == undecomposed ){
                                var[i] = decomposed;
                                decomposeT ( f_x, i );
                                var[i] = undecomposed;
                        }
                }
        }

      g_x = subfuncr(functpt, variable);

        if ( g_x not trivial ){
        for ( i = 0; i < literals; i++ ){
                if ( var[i] == undecomposed ){
                        var[i] = decomposed;
                        decomposeT ( g_x, i );
                        var[i] = undecomposed;
                }
        }
        }
}
```

The recursion function *decomposeT(functpt, variable)* searches through all possible choices of decomposition. In other words, it generates all possible combinations of different variable orders for the decomposition.

subfuncl(functpt, variable) is as the same as the one in *decompose(functpt)*.

subfuncr(functpt, variable) is as the same as the one in *decompose(functpt)*.

Figure 13 shows the decomposing tree for a three variable input function.

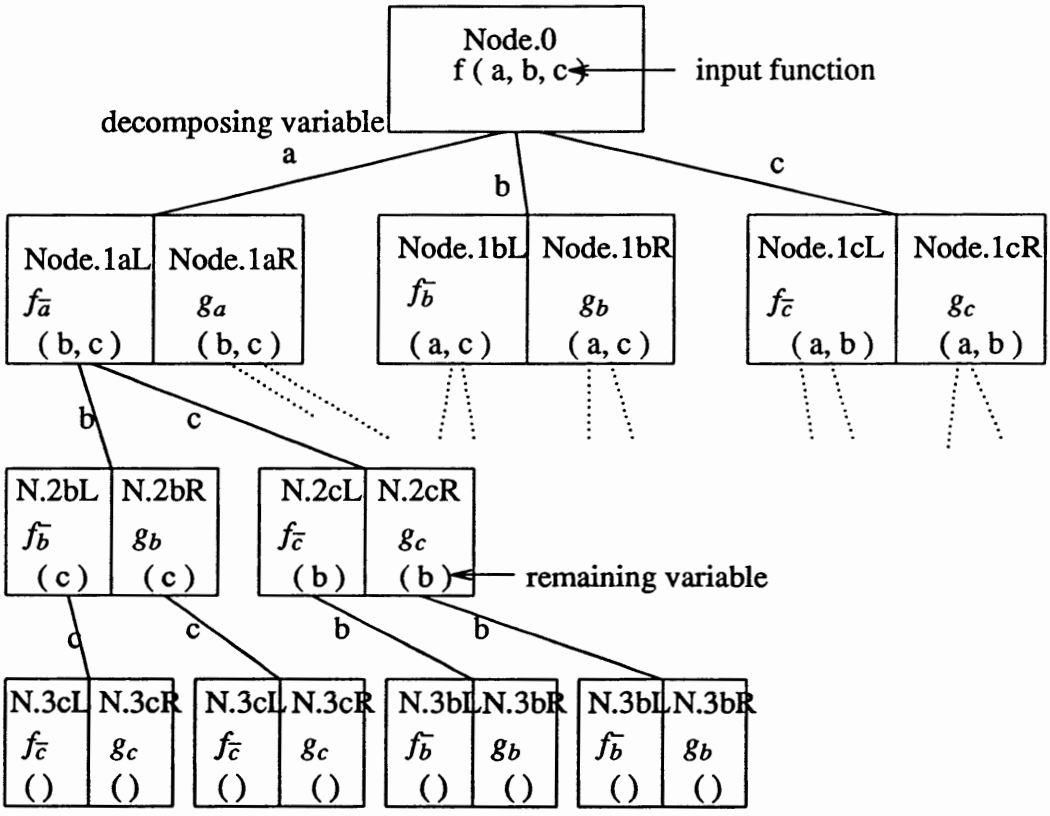


Figure 13. Decomposing tree.

For the purpose of easy explanation, we define some terms:

After decomposing variable x_i, the function will be divided into two subfunctions: ($f_{\bar{x}_i}$) and ($f_{x_i} \oplus f_{\bar{x}_i}$) or (g_{x_i}), denoted as :

L -- ($f_{\bar{x}_i}$).

R -- (g_{x_i}).

'Node.xyz'-- Letter x indicates which tree level this node is located at. Letter y indicates which variable was decomposed to create this node. Letter z indicates which part, L or R the node belongs to .

Open -- It means that the Node still contains at least an undecomposed variable, further decomposition should be performed on this Node.

Closed-- The Node contains no more undecomposed variables, no further operation on this Node is required. Node is a leaf of the tree. It contains value zero or value one.

Example IV.1 shows the decomposition of a function according to the whole tree search algorithm indicated in Definition IV.1. Suppose the input function has three variables : *a, b, c* (see figure 13).

At root, only one node -- Node.0 (the function contains variables *a, b and c*).

After decomposing variable *a* from Node.0, two new nodes are created: Node.1aL and Node.1aR (contain variable *b* and *c*). The condition of Node.1aL and Node.1aR is *Open*. Further decompositions should operate on them.

If decomposing variable *b* instead of *a* from Node.0, two other nodes will be created : Node.1bL and Node.1bR (contain variable *a* and *c*). Condition: *Open*.

If continue decomposing variable *b* from Node.1aL, the children nodes are: Node.2bL and Node.2bR (contain only variable *c*). Condition: *Open*.

Node.2bL and Node.2bR contain only one variable -- variable *c*. Further decomposition could be only decomposing variable *c*. No other choice. The children are Node.3cL and Node.3cR.

Node.3cL, Node.3cR contain no variable, they are the leaves of the tree. No further operation on them is possible. The condition of Node.3cL and Node.3cR is *Closed*. No further operation on them is possible.

The exact searching algorithm creates the decomposing tree which shows all possible decompositions. As mentioned before, the order of decomposing variables at each level determines the circuit structure. The decomposing tree illustrates all different ways to decompose an input function, however, for reaching the best solution, more methods

need to be used.

Definition IV.2: PRM Tree -- The tree describing the circuit realization obtained from the decomposition tree.

Figure 14 shows a PRM tree for a three variable input function.

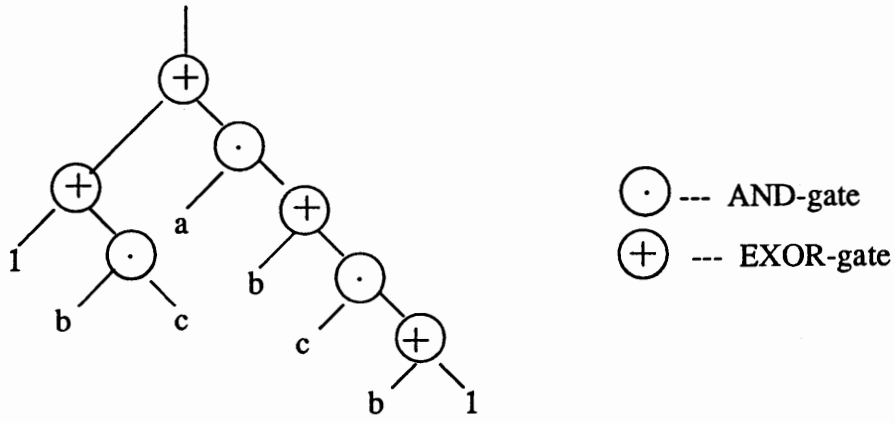


Figure 14. PRM tree circuit.

As mentioned before, after using Davio Expansion form, $f = f_{\bar{x}_i} \oplus x_i \cdot g$ to decompose a variable x_i from a given function, two subfunctions, $f_{\bar{x}_i}$, g_{x_i}, and an AND-EXOR gate structure are created. Their relationship is indicated in Figure 15.

From now on, "GN"-- Gate-Node, will be defined to store the gate structure. There are some notations:

(1) The gate structure depends on the condition of the L($f_{\bar{x}_i}$) and R(g_{x_i}). For example, the structure of GN.1x_i is decided by Node.1x_iL and Node.1x_iR.

(2) We use *GN.xy* to label each Gate-Node, where x indicates which tree level this node ls located at. Letter y indicates which variable was used in the decomposition to create this node.

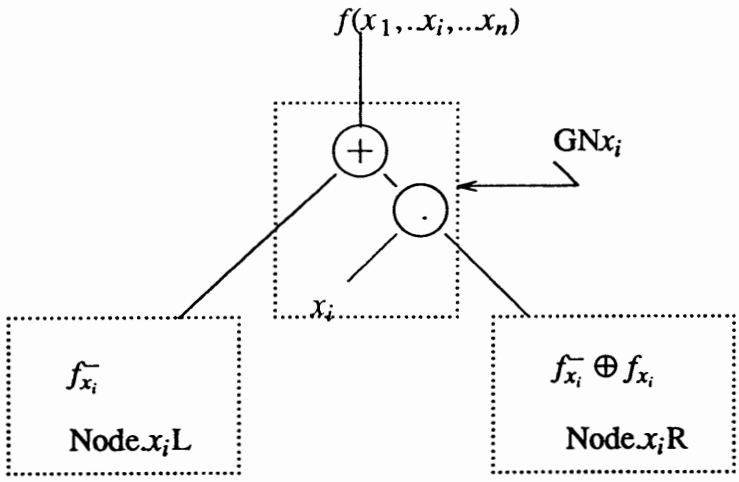


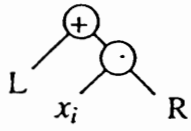
Figure 15. Relationship between gate structure and expansion subfunctions.

(3) After decomposing with respect to all variables from all subfunctions, the connection of all the gate structures constructs the PRM Tree. In other words, the PRM Tree is built up of the GNs.

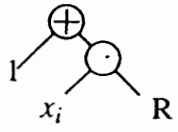
(4) The contents of GN could be: (i) an AND-gate, (ii) an EXOR-gate, (iii) an AND-gate and an EXOR-gate (like the one in Figure 15). (iiii) value 1 or 0.

How do the L and R decide the gate structure of a GN? The rules are as follows:

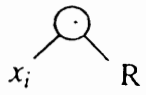
(1) If Node.L = Open, Node.R = Open

then 

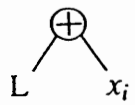
(2) If Node.L = 1, Node.R = Open

then 

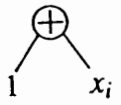
(3) If Node.L = 0, Node.R = Open

then 

(4) If Node.L = Open, Node.R = 1

then 

(5) If Node.L = 0, Node.R = 1

then 

(6) If Node.L = 0, Node.R = 1

then x_i

(7) If Node.L = 1, Node.R = 0

then 1

(8) If Node.L = Open, Node.R = 0

then L

L -- The GN formed from the NodeL.

R -- The GN formed from the NodeR.

The structure of the PRM Tree is based on the structure of the Decomposing Tree. The PRM Tree is constructed while the Decomposing Tree is generated. The searching method for the Decomposing Tree is "Depth-first", and the way to built the PRM Tree is "from bottom up".

When the first time reaching the leaf of the Decomposing Tree during the Decomposing Tree generation, the PRM Tree starts to be created. In this way, during the process of PRM Tree and Decomposing Tree creation, the program could always compare and choose the better GN. It provides then the best circuit structure. Before reaching the leaf

of the Decomposing Tree, one couldn't decide which GN is better, because the decomposing process continues in the current "Node", the Decomposing Tree is still growing deeper, the gate structure under each possible decomposing way is unknown.

Let's refer to Figure 13 and Figure 15. When decomposing variable *a* in Node.0, Node.1aL and Node.1aR are created. According to the condition of Node.1aL and Node.1aR, structure GN.1a is generated. Same, if decomposing variable *b* in Node.0, Node.1bL, Node.1bR and structure GN.1b are generated.

At this point, even if the structures of GN.1a and GN.1b are known, one can not determine which variable should be chosen to decompose to obtain a better result (i.e. the PRM Tree having less number of gates). This is because the gate structure under nodes: Node.1aL, Node.1aR, Node.1bL and Node.1bR are still unknown.

While creating the PRM Tree from the bottom up, in each GN structure the total gate number under this GN is already known. It makes possible to determine which GN should be chosen, and which GN should be deleted.

If in a particular node of the Decomposing Tree, the function contains only one variable, than only one possible decomposition exists for this node. Therefore only one GN will be created. If there are more than one decomposing choices at a particular node, different GNs will be generated under this node for different decomposing variables. In this case, the program will compare all GN's to find out the GN which has the minimum number of gates, in order to determine which variable should be chosen for the decomposition at this node.

Figure 16 shows part of a Decomposing Tree and part of a PRM Tree for a three input variable function. Using Figure 16, we could explain some basic steps of generating the PRM Tree.

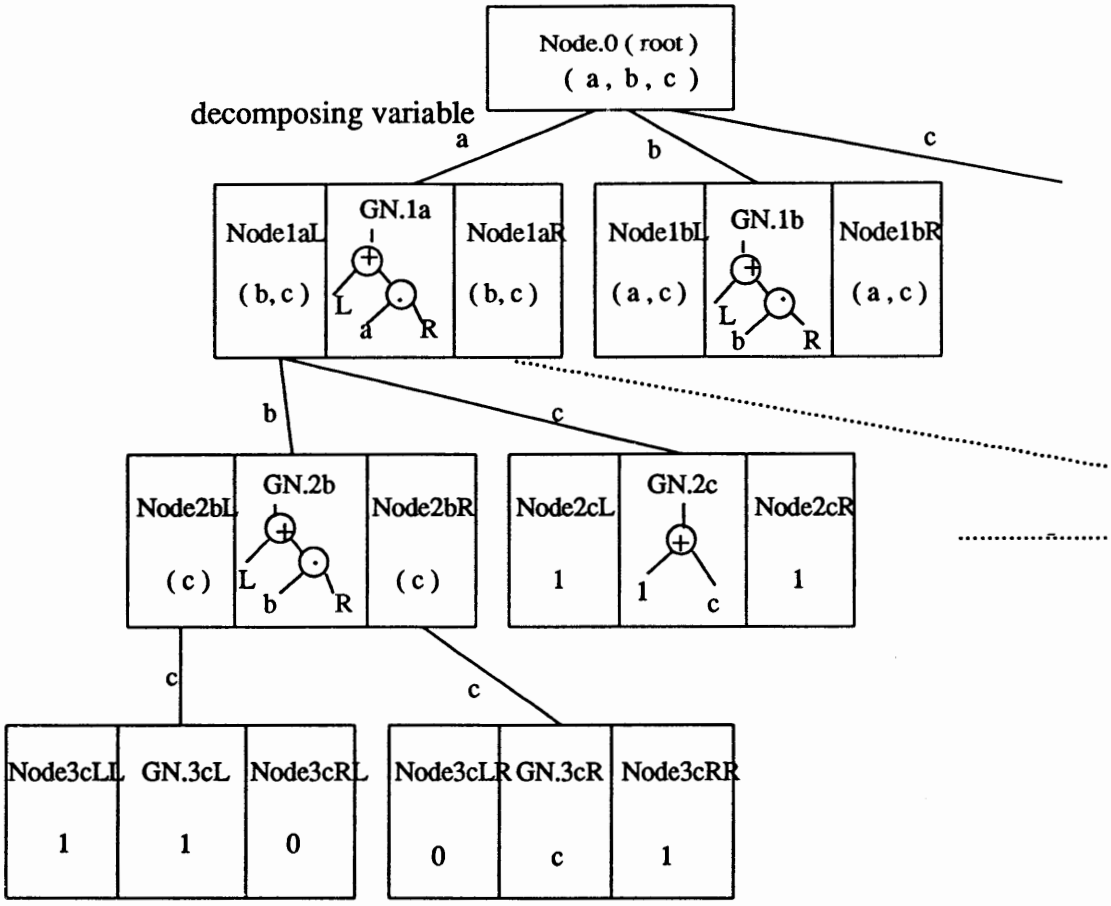


Figure 16. Relationship of the decomposing tree and the PRM tree.

The generating of PRM Tree starts from the bottom. Here, the generating order of the GNs is:

(1) GN.3cL. (2) GN.3cR. (3) GN.2b.

Linking the L of GN.2b to GN.3cL and the R of Gn.2b to GN.3cR.

(4) GN.2c. Comparing GN.2b with GN.2c and storing GN.2c (Because here GN.2c includes less number of gates than GN.2b does). In the case when the obtained GN contains no gate, the program will not continue to search the other possible GNs for the equivalent position, because it could not find a GN which is better than this one which contains no gate. For example, here, if GN.2b contains no gate, the program would not generate GN.2c but just save GN.2b and connect it

to the L of the GN at one level up the tree (here, the up-level GN is GN.1a).

(5) Generate the GN under Node.1aR, let's say GN.2b.

(6) Generate GN.1a. Link L and R of GN.1a to GN.2c and GN.2b.

(7) Generate GN.1b, compare GN.1a with GN.1b, store the better one, and so on.

So, one could see that the program checks all possible cases and provides the best PRM Tree.

IV.1.3 Example for the Exact PRM Tree Searching

The input function for the program is in the form of ESOP (Exclusive Sum of Product). For example, a three variables function:

$$f(a,b,c) = \bar{a}\,\bar{c} \oplus \bar{a}\bar{b}c \oplus abc \oplus a\bar{b}\bar{c}$$

will be presented in the following form:

```
.i 3
.o 1

0-0
001
111
100
.e
```

i -- the number of inputs.

o -- the number of outputs.

"1" -- the variable is in positive polarity.

"0"-- the variable is in negative polarity.

"-" -- the variable is not in the cube.

Each column corresponds to a variable. Each row is one cube. Such as,

"0-0" represents cube $\bar{a}\,\bar{c}$,

"001" represents cube $\bar{a}\bar{b}c$, etc.

When decomposing a variable, such as a, the subfunctions $f_{\bar{a}}$ and ($f_a \oplus f_{\bar{a}}$) are created. The program will check all cubes of the input function to decide which cube belongs to which subfunction ($f_{\bar{a}}$ or ($f_a \oplus f_{\bar{a}}$)) or belongs to both parts. Mark '-' would be used to indicate the decomposed variable. According to the definition of:

$$f_a = f(a = 1)$$

$$f_{\bar{a}} = f(a = 0)$$

For the above function,

$f_a = bc \oplus \bar{b}\bar{c}$ (From terms: abc, $a\bar{b}\bar{c}$)

$f_{\bar{a}} = \bar{c} \oplus \bar{b}c$ (From terms: $\bar{a}\,\bar{c}$, $\bar{a}\bar{b}c$)

$f_a \oplus f_{\bar{a}} = bc \oplus \bar{b}\bar{c} \oplus \bar{c} \oplus \bar{b}c$

One could see that the term which doesn't contain $\bar{a}$ will appear to the f_a part. Because when $a = 1$, $\bar{a} = 0$, the term which contains $\bar{a}$ is zero. The same reason, the term which doesn't contain a will appear to the $f_{\bar{a}}$ part. If variable a appears in the term as a "-" (don't care), this term then belongs to both f_a and $f_{\bar{a}}$, because doesn't matter if $a = 1$ or $a = 0$, this term will not be zero. Also, the term which contains a as a "-" (don't care) will not be in $f_a \oplus f_{\bar{a}}$, since it appears twice (from f_a and $f_{\bar{a}}$)and gets canceled.

So the cubes in which a = "0" or a = "-" will be selected to the $f_{\bar{a}}$ part while the cubes in which a = "0" or a = "1" will be selected to the $f_a \oplus f_{\bar{a}}$ part. In the above example, cubes "0-0" and "001" belong to $f_{\bar{a}}$ part, cubes "0-0", "001", "111"and "100" belong to ($f_a \oplus f_{\bar{a}}$). So, after decomposing variable a from the input function, $f_{\bar{a}}$ and ($f_a \oplus f_{\bar{a}}$) are created. Variable a will not appear in the subfunctions any more. The position of variable a is replaced with "-". So

$f_{\bar{a}}$ includes: "--0" and "-01".

($f_a \oplus f_{\bar{a}}$) includes: "--0", "-01", "-11" and "-00".

Variable a is no longer in the cubes. Now $f_{\bar{a}}$ and ($f_a \oplus f_{\bar{a}}$) still contain variables b and c, further decomposition should be performed on them. The decomposing process continues until no more variables remain in the cubes. In other words, the decomposing terminates when the cubes contain only the "-"s. For example, here, if $f_{\bar{a}}$ contains only cube "---", then no further decomposition will be applied to $f_{\bar{a}}$. As the same, if only cube "---" is in ($f_a \oplus f_{\bar{a}}$), no Decomposing Tree will grow under ($f_a \oplus f_{\bar{a}}$). Use the previous definition:

If only cube "---" appears in the "Node", this "Node" is *Closed.* Otherwise, the "Node" is *Open.* If the Node only contains one "---", the value of this Node is 1. If the Node contains nothing, the value of this Node is 0.

Now, we use the above function as an example to explain the Exact PRM Tree searching (See Figure 17).

Figure 17 shows that, at Node.0, there is the input function:

$$f(a,b,c) = \bar{a}\,\bar{c} \oplus \bar{a}\bar{b}c \oplus abc \oplus a\bar{b}\bar{c}$$

Node.0 contains: 0-0, 001, 111 and 100.

Check the condition of Node.0 -- *Open.* Decomposing variable a from "Node.0", generate N1aL1 and N1aR2.

N1aL1 ($f_{\bar{a}}$): --0, -01.

N1aR2 ($f_a \oplus f_{\bar{a}}$): --0, -01, -11, -00.

Check the condition of N1aL1 -- *Open.* Decomposing variable b from N1aL1, generate N2bL1 and N2bR2.

N2bL1: --0, --1.

N2bR2: --1.

Check the condition of N2bL1 -- *Open.* Decomposing variable c from N2bL1, generate N3cL1 and N3cR2.

$$f\ (a,\ b,\ c\ ) = \bar{a}\,\bar{c} \oplus \bar{a}\bar{b}c \oplus abc \oplus a\bar{b}\bar{c}$$

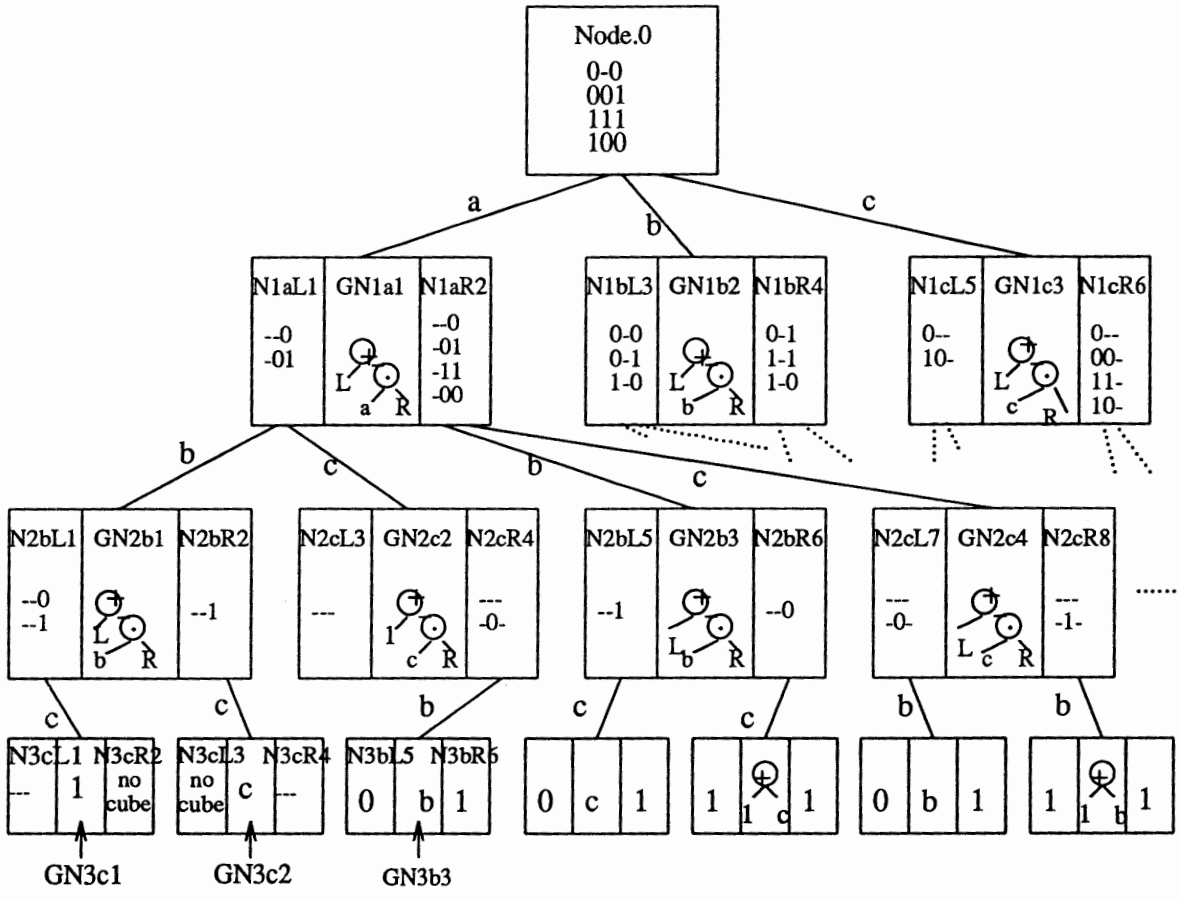


Figure 17. PRM tree generation.

N3cL1: ---.

N3cR2: no cube.

Check the condition of N3cL1 -- *Closed.* The property of N3cL1 : 1.

Check the condition of N3cR2 -- *Closed.* The property of N3cR2 : 0.

According to the properties of N3cL1 and N3cR2, generate GN.3c1.

GN.3c1: value 1.

Check the condition of N2bL1 -- *Closed.* The property of N2bL1 : function.

Check the condition of N2bR2 -- *Open.* Decomposing variable *c* from N2bR2, generate N3cL3 and N3cR4.

N3cL3: no cube.

N3cR4: ---.

Check the condition of N3cL3 -- *Closed.* The property of N3cL3 : 0.

Check the condition of N3cR4 -- *Closed.* The property of N3cR4 : 1.

According to the properties of N3cL3 and N3cR4, generate GN.3c2.

GN.3c2: variable *c*.

Check the condition of N2bR2 -- *Closed.* The property of N2bR2 : function.

According to the properties of N2bL1 and N2bR2, generate GN.2b1. The *L* in GN.2b1 links to GN.3c1 and the *R* in GN.2b1 links to GN.3c2.

GN.3c2:

(1) an AND-gate -- It takes variable *b* and *c* as its two inputs.

(2) an EXOR-gate -- It takes value 1 and the output of the AND-gate as its two inputs.

Check the condition of N1aL1 -- *Open.* Decomposing variable *c* from N1aL1, generate N2cL3 and N2cR4.

N2cL3: ---.

N2cR4: ---, -0-.

Check the condition of N2cL3 -- *Closed.* The property of N2cL3 : 1.

Check the condition of N2cR4 -- *Open.* Decomposing variable *b* from N2cR4, generate N3bL5 and N3bR6.

N3bL5: no cube.

N3bR6: ---.

Check the condition of N3bL5 -- *Closed.* The property of N3bL5 : 0.

Check the condition of N3bR6 -- *Closed.* The property of N3bR6 : 1.

According to the properties of N3bL5 and N3bR6, generate GN.3b3.

GN.3b3: variable *bR*.

Check the condition of N2cR4 -- *Closed.* The property of N2cR4 : function.

According to the properties of N2cL3 and N2cR4, generate GN.2c2. The *R* of GN.2c2 links to GN.3b3.

GN.2c2:

(1) an AND-gate -- It takes variable *b* and *c* as its two inputs.

(2) an EXOR-gate -- It takes value 1 and the output of the AND-gate as its two inputs.

Comparing GN.2b1 with GN.2c2, since the nodes have the same cost (the same number of gates), then, either one could be stored as a subPRM Tree for the up level GN. In the program, if both GNs has the same cost, the first one will be saved. So here, GN.2b1 is saved and GN.2c2 is deleted. If during the comparison, the cost of the two GNs is different, the program will store the one which has the smaller cost and will delete the other one.

Check the condition of N1aL1 -- *Closed.* The property of N1aL1 : function.

Check the condition of N1aR2 -- *Open.*

Repeat the same processes which was operated on N1aL1, until the condition of N1aR2 is *Closed.* The property of N1aR1 : function.

Then generate GN.1a1 according to the properties of Node.1aL1 and N1aR2. The *L* of GN.1a1 links to GN.2b1 and the *R* of GN.1a1 links to the GN which is obtained from N1aR2. In this example, the *R* of GN.1a1 links to GN.2b3.

Back up one level of the tree. The condition of Node.0 is *Open.* Another variable such as *b* will be chosen for the decomposition. The previous processes (creating Nodes, generating GNs, comparing and recording the better GN) will repeat until all variables in Node.0 have been decomposed.

When the condition of Node.0 is *Closed*, the whole search is completed. The output is a tree of the GNs. That is the PRM Tree. It gives the best AND-EXOR gates connection for realizing the requested Boolean function.

The output of the program could be presented in three equivalent forms: formula, graph and BLIF form. The formula form presents the result in a standard way. The BLIF form is necessary for interfacing PROMPT to U.C. Berkeley tools which allows, among other, to verity the correctness of the produced by it results. The graph form makes the result easier to read.

The obtained output of our example is:

(1) formula:

f = [[[1] + [[b] * [c]]] + [[a] * [[c] + [[b] * [[1] + [c]]]]]]

(2) BLIF form (the corresponding diagram is Figure 18):

```
.inputs a b c
.outputs m0
.names m1 m3 m0
01 1
10 1
.names m2 m1
0 1
.names b c m2
11 1
```

```
.names a m4 m3
11 1
.names c m5 m4
01 1
10 1
.names b m6 m5
11 1
.names c m6
0 1
.end
```

(3) graph :

```
+
  *
    +
      *
        +
          c
          1
        b
      c
    a
  +
    *
      c
      b
    1
```

* -- AND-gate, + -- EXOR-gate.

When the above graph is rotated 90 degree clock-wise and the connection lines are added, one obtains the PRM Tree (Figure 14).

Actually, the tree structure is equivalent to a circuit. One can convert PRM Tree into a circuit very easily. The corresponding circuit is presented in Figure 18.

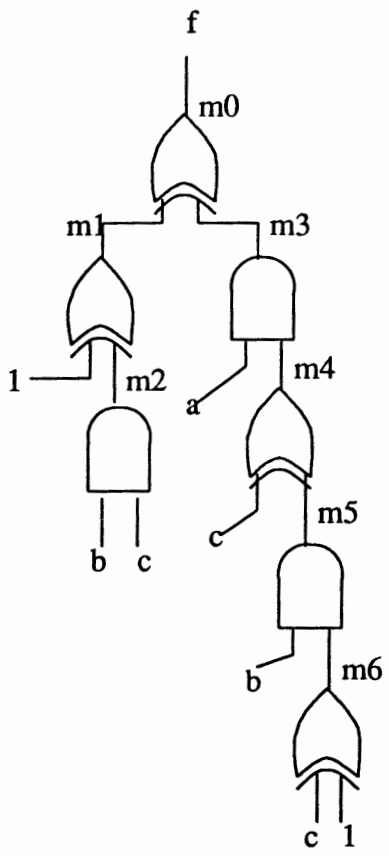


Figure 18. PRM circuit.

IV.2 HEURISTIC PRM TREE SEARCHING

IV.2.1 Basic Method

The execution time of complete tree searching programs is large for problem instances of high complexity. Thus, the PRM Tree searching method presented in the previous section is applicable to only very small problems. Therefore, even though the Exact PRM Tree searching allows to obtain the best solution, no solution for functions depending on many input variables can be generated in a realistic time. Therefore, we need to find some methods that will produce solutions in a reasonable time.

As could be observed in the previous section the size of the Decomposing Tree can be extremely large when the function has a large number of input variables. Therefore, searching the whole solution space takes a large amount of time. In order to reduce the search time, the solution space has to be reduced. One of the methods to achieve this is to limit the choices of the possible decompositions for a function.

At the root node of the tree for a function of n input variables there are n decomposition choices. If one allows only one variable to be selected for the decomposition at this level, the size of the Decomposing Tree could be reduced exponentially. Therefore, to reduce the time of the tree searching process, cutting off a branch at the top level of the tree is much more effective than cutting off a branch at a lower lever of the tree. But it often leads to an nonoptimal result.

The next section illustrates a heuristic to decrease the solution space to obtain results in a reasonable amount of time, even for large functions.

IV.2.2 Algorithm

To reduce the solution space for a large function to a space that is computational feasible, the heuristic PRM Tree searching algorithm allows only one decomposition choice in the top levels of the Decomposing Tree. That is, in the top levels that are specified by the user, the decomposition is performed only for one selected variable. In this way, the tree searching time can be reduced tremendously. The heuristic for the variable selection used is given by: the variable that occurs most often in the cubes is selected. In order to obtain results as close as possible to the exact solution, at certain level of the Decomposing Tree, the program starts to checking all possible variable decompositions.

The algorithm of the heuristic PRM Tree searching program is given by function *hprms()*. The function *hprms(functpt, selectvar, level)* is recursive. It accepts three arguments: *functpt, selectvar* and *level*: *functpt* is a pointer which points to the input function,

selectvar is the variable selected for the decomposition, *level* is the number of top levels of the Decomposing Tree for which the decomposition is only performed for the variable selected according to the above mentioned heuristic.

The characteristic of function *hprms()* is: during the Decomposing Tree generation, at the top levels, only one variable is chosen for the decomposition under each function. After those top levels, all possible decomposition choices will be checked.

The pseudo-code for the function *hprms()* is as follows:

```
hprmt ( functpt, selectvar, level )
int level;
{
        level++;

        f_x = subfuncl ( functpt, selectvar );
        if ( f_x not trivial )
        {
                if ( level < specified number ) /* user set the number */
                {
                        var = pick_var ( f_x );
                        f = hprmt ( f_x, var, level );
                }
                else
                        /* the exact tree searching */
                        f = decomposeT ( f_x );
        }

        g_x = subfuncr ( function, variable );
        if ( g_x not trivial )
        {
                if ( level < specified number ) /* user set the number */
                {
                        var = pick_var ( g_x );
                        g = hprmt ( g_x, var, level );
                }
                else
                        /* the exact tree searching */
                        g = decomposeT ( g_x );
        }
}
```

The routines in the function *hprmt(functpt, selectvar, level)*:

pick_var(function) -- Select a variable for the decomposition according to the heuristic rule. The rule used: the variable that occurs most often in both positive and negative polarities in the cubes is selected.

subfunctionl(functpt, selectvar) -- compute the subfunction of the input function which corresponds to $f_{\overline{x}_i}$.

subfunctionr(functpt, selectvar) -- compute the subfunction of the input function which corresponds to $g_{x_i} = f_{\overline{x}_i} \oplus f_{x_i}$.

decomposeT(function) -- Check all possible decompositions and record the best one. This is the same function mentioned in the previous section.

f is trivial if f $\in$ { 0, 1, x, $\overline{x}$ }.

Figure 19 shows the Decomposing Tree generated by *hprmt(functpt, selectvar, level)*.

Figure 19 illustrates the following issues:

The root of the tree is given by Node0. Node0 stores the input function. Even though the function in Node0 contains n variables $x_1, x_2, ...x_i, ..x_n$, only one variable will be selected for the decomposition at Node0. Here, we assume that x_1 is the selected variable. After the decomposition, two nodes: Node1x_1L and Node1x_1R are generated.

At the second level, there are two Nodes: Node1x_1L and Node1x_1R. Node1x_1L stores the cubes which correspond to the subfunction $f_{\overline{x_1}}$. Node1x_1R stores the cubes which correspond to the subfunction g_{x_1}. Both nodes contain many undecomposed variables (from x_2, x_3 to x_n).

The decomposing process is performed as before: only one variable will be selected for the decomposition at each node. For example here, decomposing x_2 at Node1x_1L, the nodes Node2x_2L and Node2x_2R are created. Decomposing x_3 at

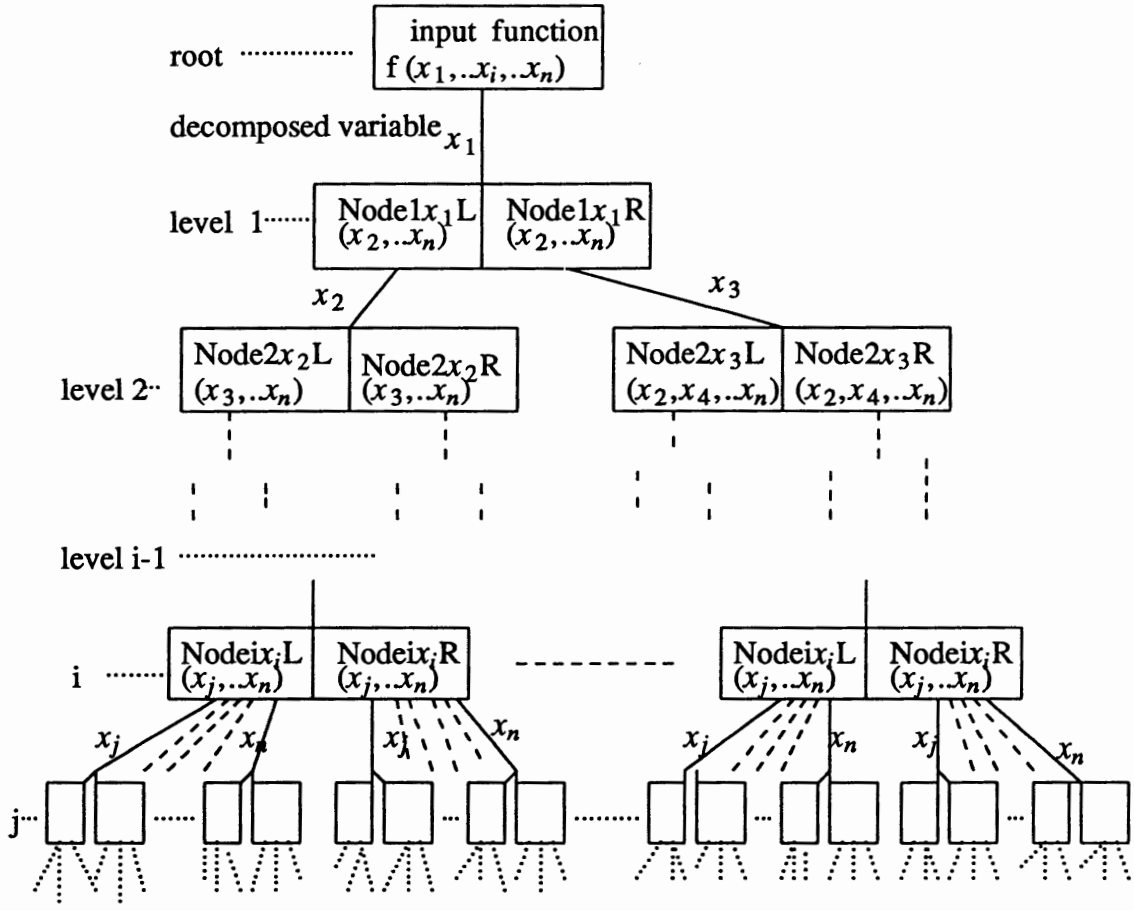


Figure 19. Diagram for the heuristic tree searching.

Node1x_1R, the nodes Node2x_3L and Node2x_3R are created.

Continuing the decomposition for all remaining Nodes only one decomposition choice is allowed for each Node, until reaching the level where the *exact* searching starts.

For example, suppose level i is the level where to start checking all possible decompositions at each node. Starting from level i, as many not yet decomposed variables a Node contains, that many decomposing choices have to be checked for this Node.

For example, Nodeix_iL contains variables $x_j, x_k, \ldots x_n$. Then, the program will check all possible decomposition branches, decomposing x_j, decomposing $x_k, \ldots x_n$, it will find the best result, and will return it to the upper level. In other words, the *exact* searching is executed on each Node at level i.

The heuristic PRM Tree searching programs allowed to reduce the computation time. However, with the trade-off of generating an nonoptimal result.

For the mapping of the obtained result to the CLi 6000 one can observe that the structure of the architecture does not ideally match the structure of the obtained circuit. In the PRM Tree, usually, at the same tree level, the expansion variables for each node are not necessarily the same. See Figure 20.

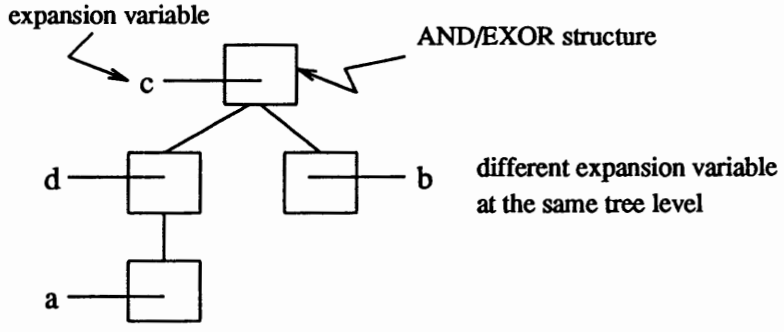


Figure 20. Diagram of PRM Tree.

Thus, for the implementation of the circuit to the CLi600, several input variables would have to be routed to cells in which they are the expansion variables. This may cause some routing inconveniences when mapping the circuit into the CLi6000 chip. Therefore, this is a trade-off between the necessary routing cells and the necessary logic cells.

For the purpose to better match the tree circuits to the CLi6000 chip, another version of the heuristic searching program called Heuristic Reed-Muller Tree Searching (

HRMT) is introduced in the next section.

IV.2.3 RM Tree

In the RM Tree, all nodes at the same tree level have the same input variable. Such a 'levelized' distribution of input variables matches closer to the local bus structure of the CLi 6000, see Figure 21 and Figure 2.

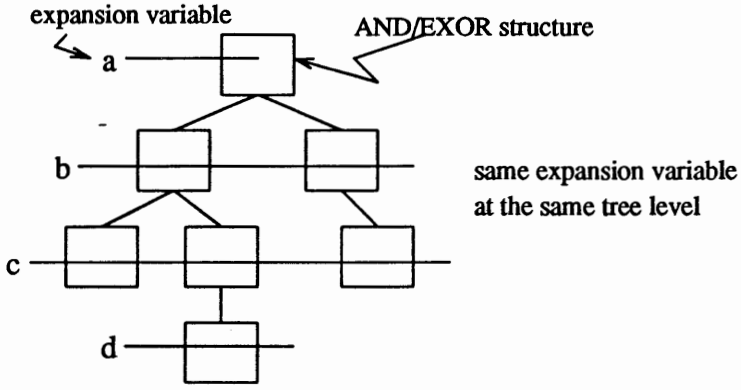


Figure 21. Diagram of RM Tree

As can be observed from Figure 21, the RM Tree provides a circuit structure that better matches the CLi6000 chip. The input variable distribution of the obtained circuit matches ideally the local bus structure provided by the CLi6000. Thus, fewer routing cells should be required to place and route the circuit.

In the HRMT searching, a decomposing variable is selected each level of the *Decomposing tree* according to a simple heuristic. Each function in the same level of the tree is decomposed with respect to this selected variable. Further improvements of the heuristic could be to find a method to predict the best decomposing variable for each level.

IV.3 THE GRM TREE SEARCHING

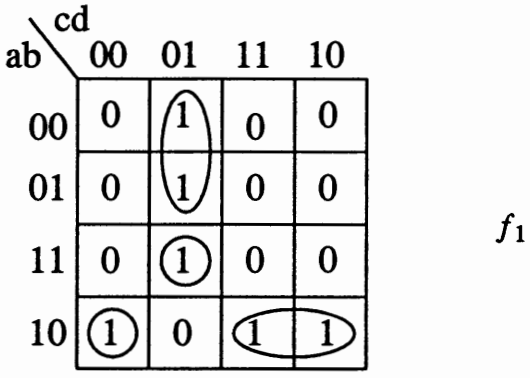
The Exact PRM Tree searching could find the best PRM circuit for a given function. If we perform some polarity transform of the original function, the program would be also to find the GRM Tree circuits. For two-level GRM forms there exists the program EQM[19] which computes the polarity that leads to the best two-level solution in an reasonable amount of time. Thus, we can apply this program as preprocessor to find a good polarity that is then used for the PRM Tree searching.

For example, we have the following function:

$$f = \bar{a}\,\bar{c}d \oplus ab\bar{c}d \oplus a\bar{b}\bar{c}\bar{d} \oplus a\bar{b}c$$

The program called EQM [19] can find the best polarity for a GRM of a Boolean function. Suppose, EQM found that the best polarity of GRM for this function is: $a,\bar{b},\bar{c},d$. The Exact PRM Tree searching assumes the polarity is *a, b, c, d.* To apply the result obtained by EQM to our PRM tree searching, we have to perform the following steps:

(i) Given is the original function f_1 (Figure 22).



f_1

Figure 22. function f_1.

(ii) Change $\bar{b}$ and $\bar{c}$ to negated variables: $\bar{b}$ --> B, $\bar{c}$ --> C. Do the following transform, generate function f_2 (Figure 23).

$\bar{b}$ ⟶ B

$\bar{c}$ ⟶ C

a	b	c	d		a	B	C	d
0	X	0	1	⟶	0	X	1	1
1	1	0	1	⟶	1	0	1	1
1	0	0	0	⟶	1	1	1	0
1	0	1	X	⟶	1	1	0	X
	f_1					f_2		

Figure 23. polarity transform for function f_1.

(iii) Take function f_2 as the input function to run the Exact PRM Tree searching (Figure 24).

aB \ Cd	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	1	1	0	1
10	0	0	1	0

f_2

Figure 24. function f_2.

Now, we find the best PRM Tree circuit for f_2. See Figure 25.

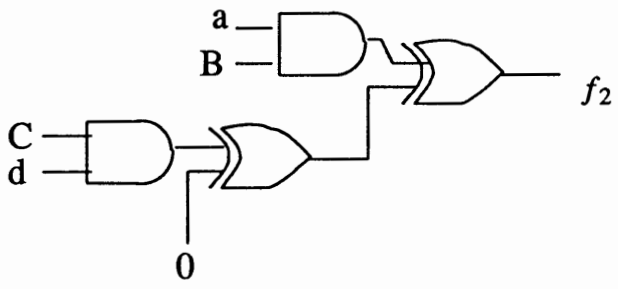


Figure 25. PRM Tree circuit for f_2.

(iv) In schematic change back: B --> $\bar{b}$, C --> $\bar{c}$. We get the GRM circuit with polarity $a\bar{b}\bar{c}d$ for function f_1. See Figure 26.

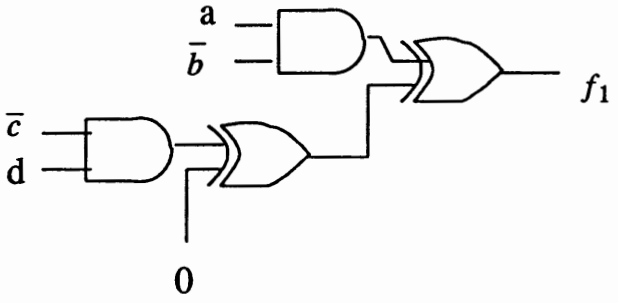


Figure 26. GRM Tree circuit for f_1.

CHAPTER V

TECHNOLOGY MAPPING

The previous chapter introduced the concept of the PRM circuit and the RM circuit. This chapter outlines the easy application of the obtained circuit to the placement and routing for the CLI 6000 FPGA series.

The basic macrocell provided by the CLI 6000 FPGA series implements a rich and powerful set of logic and routing functions. With the characteristics of the PRM circuits and the RM circuits, only certain cell states are used during the placement. Those basic logic and routing functions are showed in Figure 3.

As one can observe from Figure 3, the logic macros provided by the CLI 6000 FPGAs are identical to the logic subfunctions obtained by the introduced PRM tree searching programs. Moreover, the levelized connection of input-variables to the the PRM tree circuit matches ideally the local bus structure of the CLI 6000 FPGA series.

To illustrate the placement and routing of the different tree circuits obtained by the introduced programs to the CLI 6000 series, we show the mapping of the PRM and RM tree circuit of a standard benchmark function to the CLI 6000 series.

We selected the first output function of the con1 circuit provided by the MCNC benchmarks:

con11.tt

```
11110-
01--01
1011--
-001--
```

After the application of the RM and PRM programs we obtain the circuits shown in Figure RM and Figure PRM.

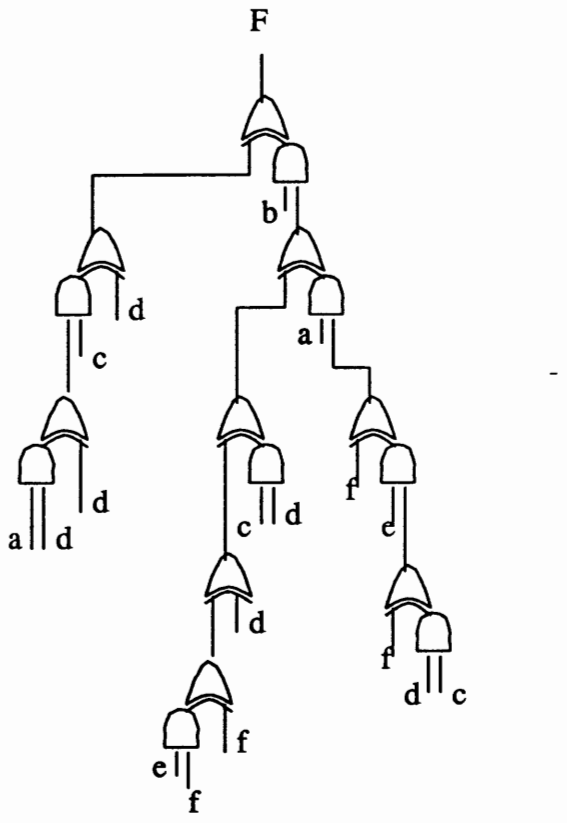


Figure 27. PRM circuit.

The placement of the PRM and RM circuits to the CLI 6000 FPGA series can be realized by the following steps:

(1) Map the nodes/gates of the circuit to functions realizable with one macrocell provided by the CLI 6000 FPGA series.

(2) Place the mapped circuit into the CLI 6000 chip.

(3) Check the connections of the macrocells between the neighboring cells. If the

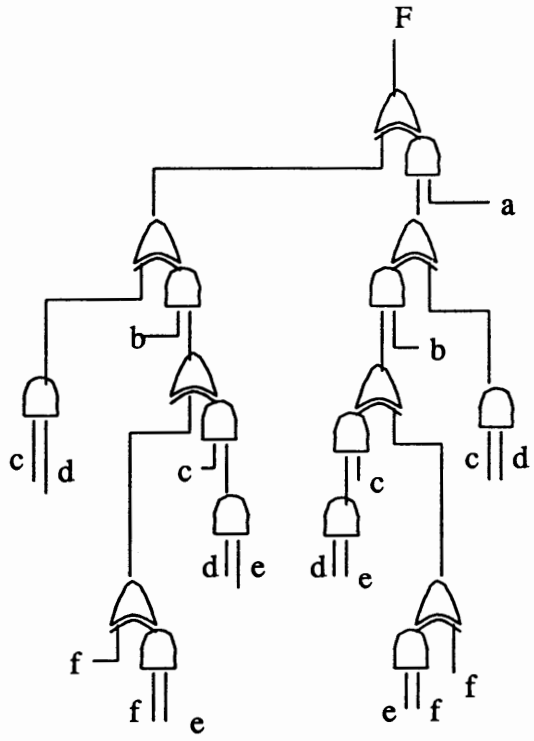


Figure 28. RM circuit.

required input or output of a macrocell can not be directly provided by the corresponding neighboring cell, an additional routing cell is needed between those two logic cells.

(4) Check the connections of each local bus to the macrocells. If the required input variable of a cell can not be provided by the local bus which is connected to this cell, one or more routing cells need to be reserved next to this cell in order to get the required input variable through another cell's local bus.

(5) Check if the number of macrocells obtained by the placement of logic and routing cells can be reduced by routing input variables directly to a particular cell using a vertical local bus.

The following graphs illustrate the placement of the PRM and RM circuit to the CLI 6000 chips.

In the pictures, each square illustrates a macrocell, the horizontal and vertical lines illustrate the local bus. Each cell connects to four local buses but can only get one input from any one of the four local buses. A and B are the inputs or outputs of the macrocells. When realizing logic functions, some logic gates in the macrocell require the A inputs while some logic gates require the B inputs, and some gates in the macrocell could provide only A outputs while some gates could provide only B outputs. So that, during the circuit placement, the input and output type (A or B) also needs to be match.

Figure 29 shows the mapping of the PRM circuit to the CLI6000 chip. One can observe that 10 logic cells and 20 routing cells are required for its implementation.

Making some input variables available on the vertical local bus and rearranging some logic cells, one obtains the PRM circuit shown in Figure 30.

The improved circuit placement takes 18 routing cells.

Figure 31 and Figure 32 show the mapping of the RM circuit to the CLI6000 chip. Because the different way of loading expansion variables to the local bus, the number of routing cells in Figure 31 is less than the one in Figure 32.

The placement of the RM circuit takes 11 logic cells and 17 routing cells.

The improved RM circuit placement needs 11 logic cells and 13 routing cells.

As we mentioned before, each macrocell is connected to four local buses (let's call them lower-bus, up-bus, left-bus and right-bus). On above placement examples, the expansion variables were loaded on only the lower-bus not the up-bus. If some chosen expansion variables be loaded also on the up-bus, it may lower the cost of the placement. Figure 33 illustrates another application of the RM circuit placement.

Figure 33 shows that only two additional loaded buses are needed, one for input variable *d* and one for *f*, instead of four, two for *d* and two for *f*, in Figure 32.

Table I shows the results of the placement. One could see that for the PRM circuit

the placement needs less logic cells but more routing cells than for the RM circuit.

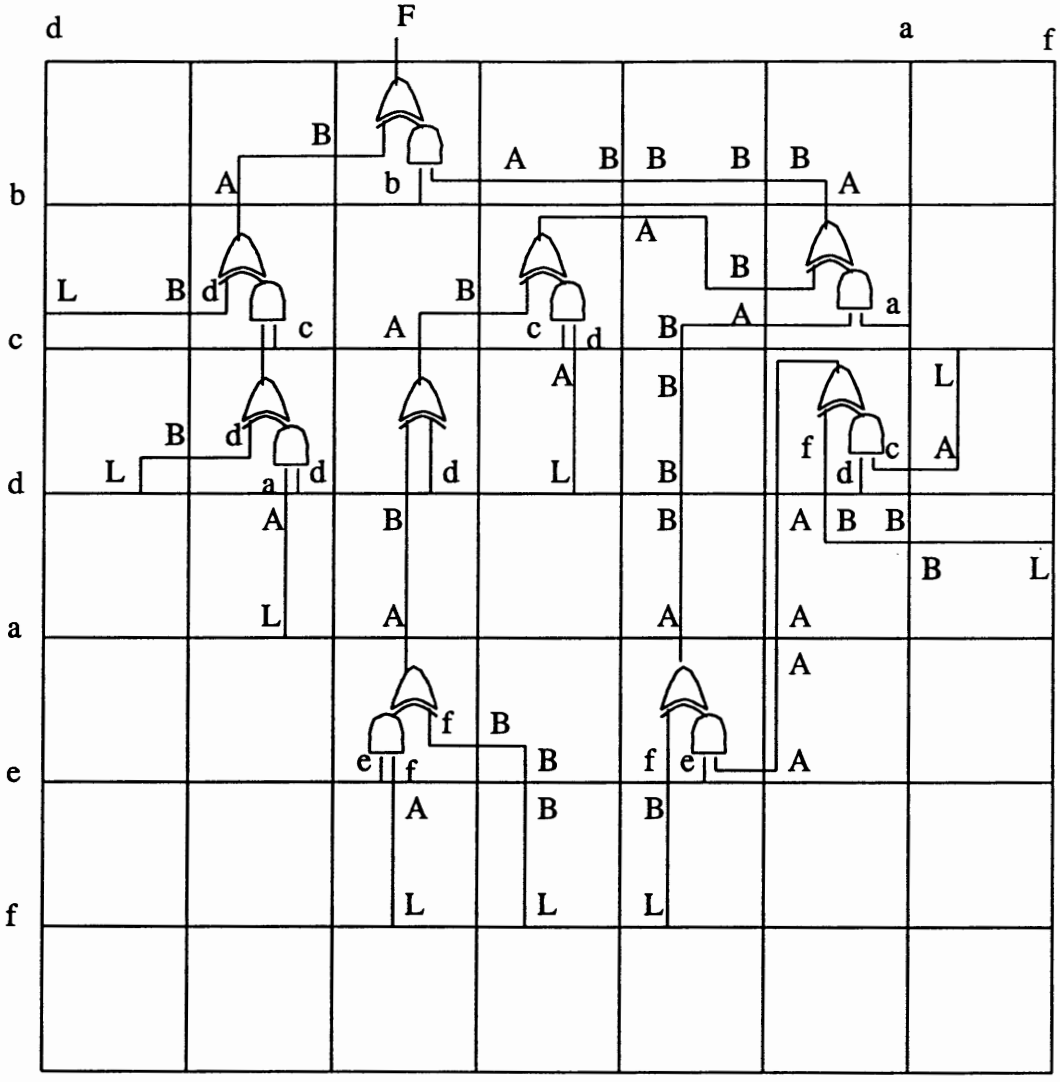


Figure 29. Mapping for the PRM circuit.

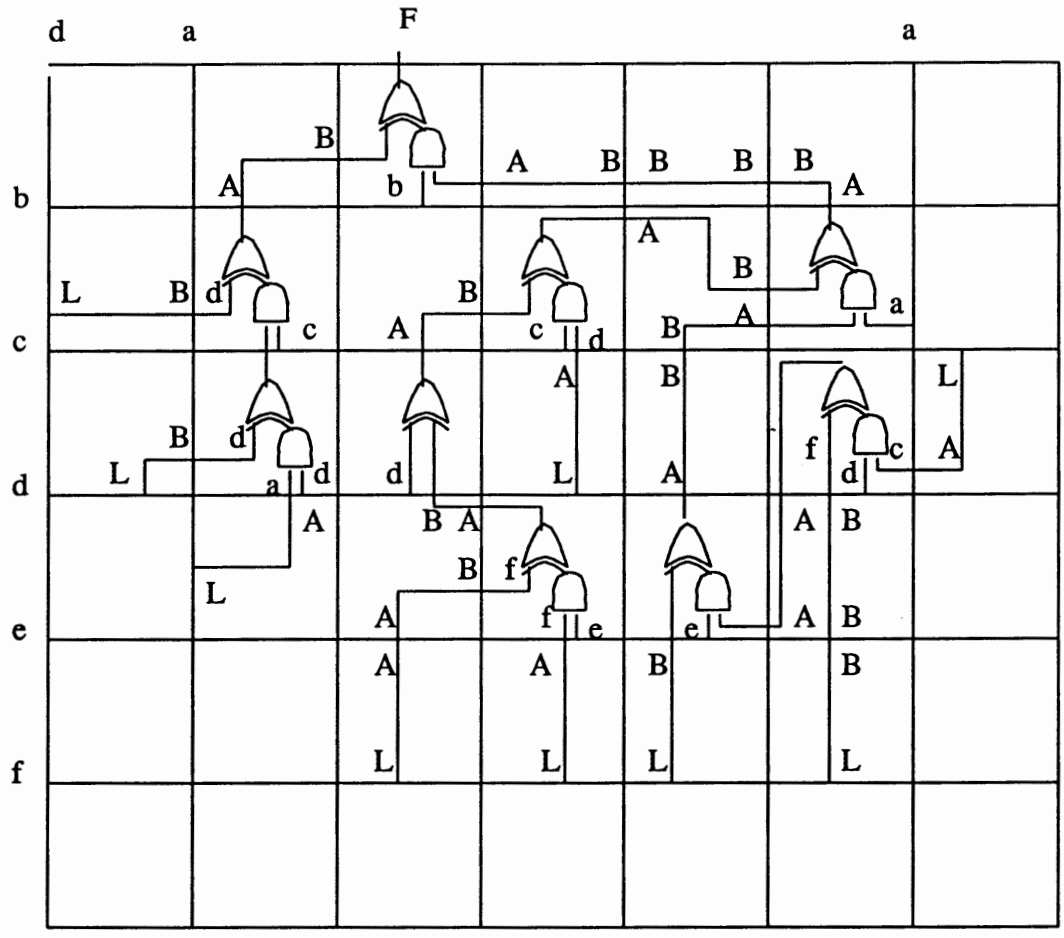


Figure 30. Mapping #2 for the PRM circuit.

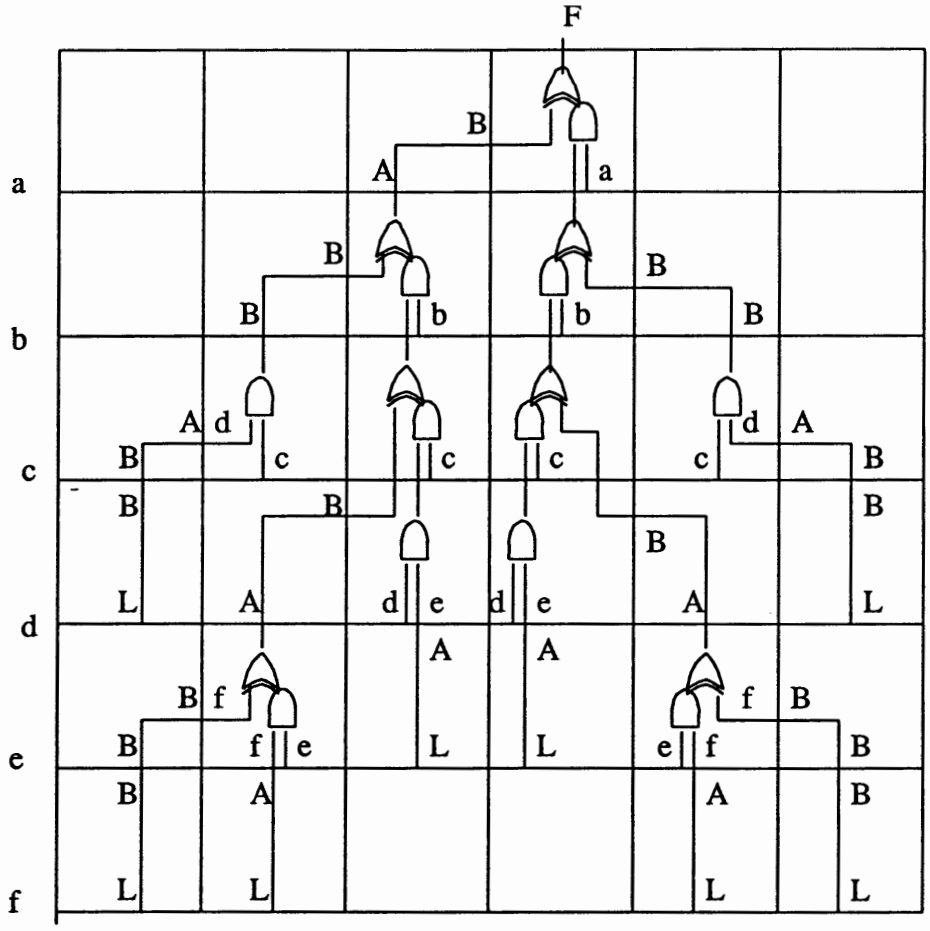


Figure 31. Mapping for the RM circuit.

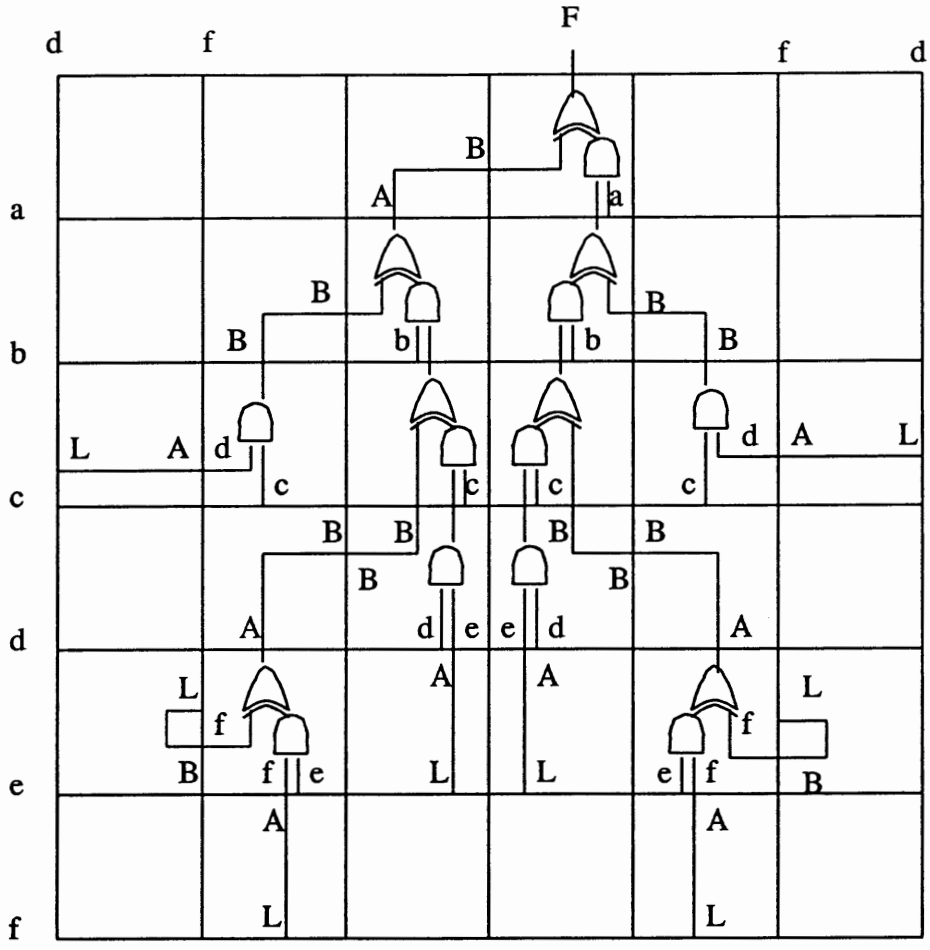


Figure 32. Mapping #2 for the RM circuit.

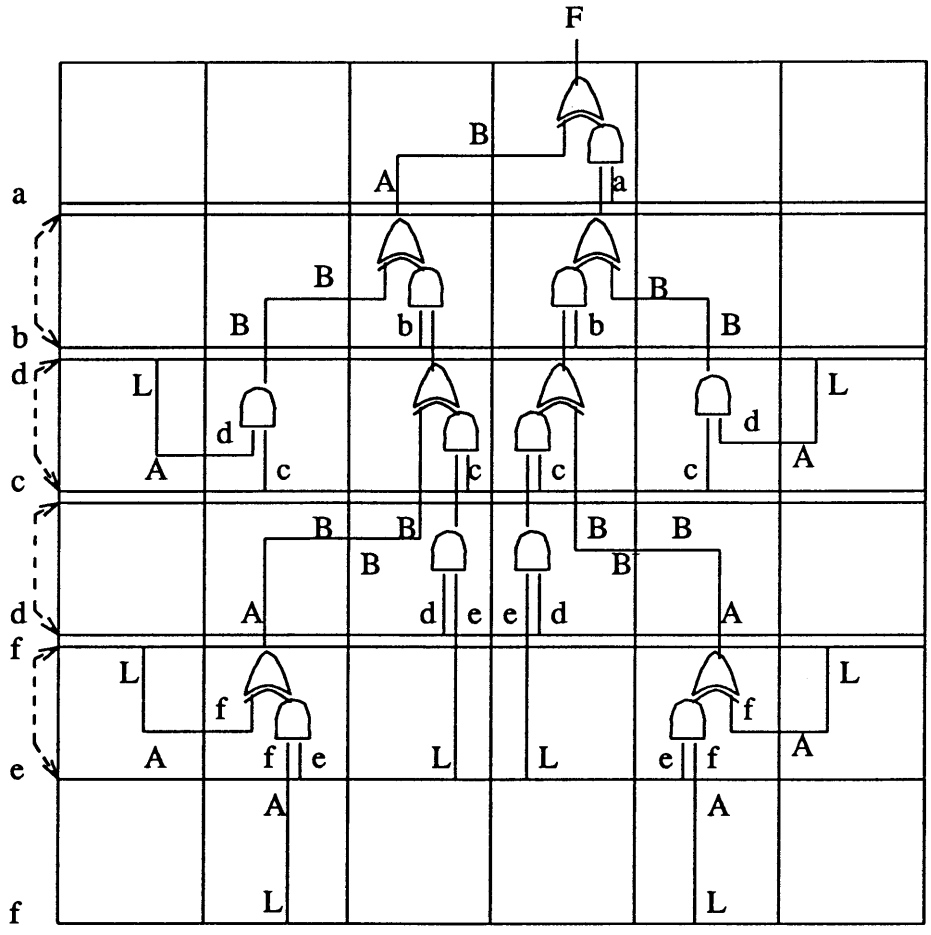


Figure 33. Mapping #3 for the RM circuit.

TABLE I

COMPARISON OF PRM CIRCUIT AND RM CIRCUIT PLACEMENT

placement#1	*# Logic cell*	*# routing cell*	*# loaded buses*
PRMcircuit	9	21	9
RMcircuit	11	17	6
placement#2	*# Logic cell*	*# routing cell*	*# loaded buses*
PRMcircuit	9	18	8
RMcircuit	11	13	10
placement#3	*# Logic cell*	*# routing cell*	*# loaded buses*
RMcircuit	11	13	8

CHAPTER VI

EVALUATION OF RESULTS

In this thesis the multiple-level Reed Muller Tree searching and the Permuted RM Tree searching have been introduced. The interest in such forms was triggered by the upcoming of the CLi6000 series of Field Programmable Gate Arrays from Concurrent Logic Inc. The CLi 6000 series provides an architecture that ideally matches the AND-EXOR structure of the RM trees. Additionally, it provides a near optimum AND-EXOR circuit to realize a given Boolean function. Therefore, it can be used in any technology that contains the AND and EXOR gates: for instance, the standard cell technology.

The multiple-level tree searching algorithms introduced in this thesis include three searching methods :

(1) Exact PRM Tree searching .

(2) Heuristic PRM Tree searching .

(3) Heuristic RM Tree searching .

The introduced exact PRM Tree searching computes the entire solution space of possible decompositions. That is, it checks all possible variable orders for the decomposition and compares the different results at each tree level. Therefore, the exact tree searching guarantees to find the PRM tree with the least number of AND-EXOR gates.

After testing a large number of functions with different numbers of inputs, we compared the results of the exact tree searching with the results of a decomposition based on a random selection of the decomposition variable. (Here we just pick a decomposition order for all variables, like *a*, then *b*, *c*, *d*....)

The test results show that, regardless how many input variables or how many input cubes, the exact tree searching always gives a better solution.

Table II shows that for realizing the same input function, the gate number obtained by the exact tree searching is always smaller than the one obtained by the random searching.

From the test results, we conclude that when the input function has less than nine input variables, the Exact PRM Tree searching is the ideal searching method. Because it always provides the best PRM AND-EXOR structure tree within a reasonable time.

The heuristic tree searching has been developed to overcome the time problem when dealing with functions having many input variables. For input functions having more than nine variables, the exact tree searching takes an extremely long time because the size of the tree increases exponentially with the number of inputs. Thus, the tree searching time increases also exponentially with the number of inputs. For example, for an eight input function, the exact tree searching may take 200 seconds, however, a nine input function would take up to eight hours. Thus, using the exact tree searching on functions having many input variables (more than nine variables) is not practical.

The presented heuristic PRM tree searching has overcome this problem. A larger number of tests proved the following facts: by using the heuristic tree searching, the searching time drops tremendously. See Figure 33. The result quality varies with different input functions. For some functions the heuristic tree searching finds the best solutions (see Table III), while for some other functions the results from the heuristic tree searching are quite different from the results of the exact tree searching. But all results are obviously in an acceptable region of the best result. Please see Figure 35.

Figure 34 illustrates the computation times for the different algorithms. It shows the searching time as a percentage of the exact tree searching time. As can be observed, the searching time for the heuristic tree searching algorithm is much less than for the

exact tree searching.

Figure 35 shows the results obtained by the two different versions of the algorithm. The cost for the AND-EXOR circuit is given by the number of AND gates and EXOR gates.

From Figure 35 one could observe that the cost obtained by the heuristic tree searching is only increased by a small amount comparing to the cost obtained by the exact tree searching.

The Figure 36 shows the searching time curve for exact tree searching and heuristic searching as a function of the number of input variables.

After testing functions having many input variables (inputs vary from 10 to 15 or even 20) by performing a heuristic tree search, the results show that the time consumption is not a problem anymore. Applying the heuristic tree searching algorithm to many functions having many input variables, it could be observed that all results have been obtained in an acceptable time. Since there are no result comparisons possible to the Exact PRM Tree searching for functions having many input variables, it is difficult to evaluate exactly how close the results provided by the heuristic PRM Tree searching are to the best results. However, we assume based on the results obtained for functions having few input variables, that the results of heuristic PRM Tree searching are also close to the exact solution.

The heuristic PRM Tree searching could be improved by finding better heuristic rules for the selection of the decomposing variables.

The RM Tree searching is developed to provide the circuits for better matching the CLi6000 chips. Because of the existence of bus connection in the CLi6000 chips, RM Tree circuit is more appropriate than PRM Tree circuit with respect to placement in CLi6000 chip. Even though the RM Tree circuits may not have the minimum number of gates (see Table IV), their property of having the same expansion variable at the same

tree level has advantage during the routing. The connectivity of the logic blocks of the CLi 6000 series matches the connectivity of the RM tree circuit. Therefore, the RM tree circuit can be directly placed. Only few additional cells for routing have to be added.

In addition, the RM Tree searching and PRM Tree searching could be extended to generate the GRM circuits. This gives us more opportunities to use the program for different applications.

The importance of our tree searching program consists in providing a new method for the logic optimization with AND-EXOR gates that are of growing industrial interest.

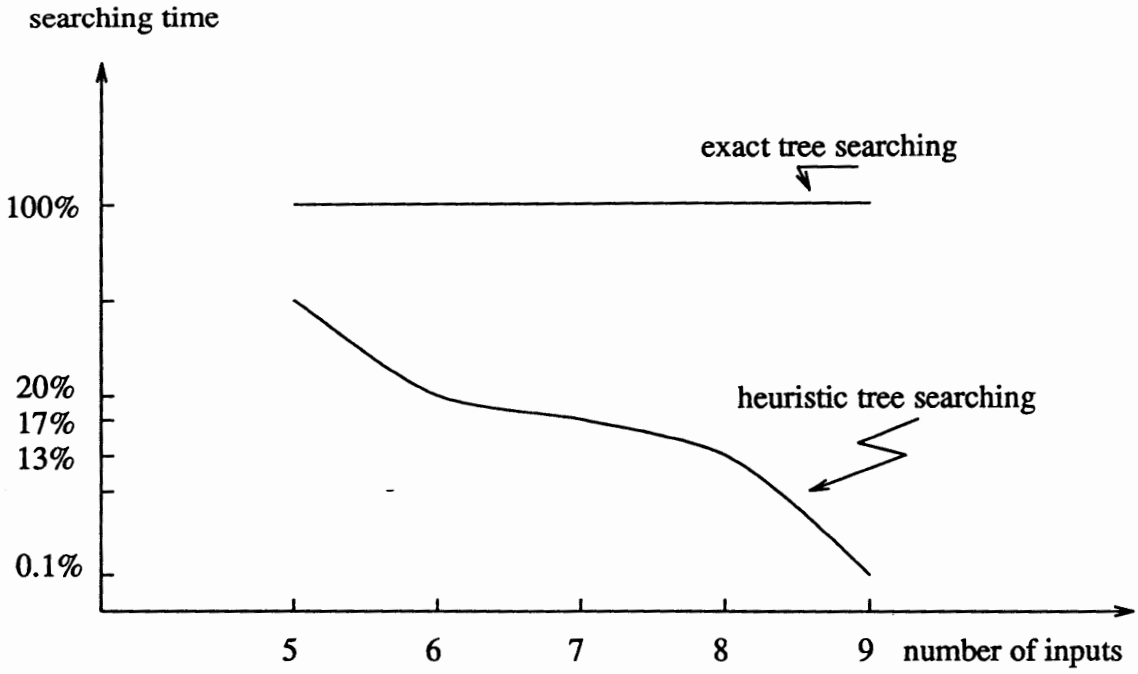


Figure 34. Heuristic searching time versus exact searching time.

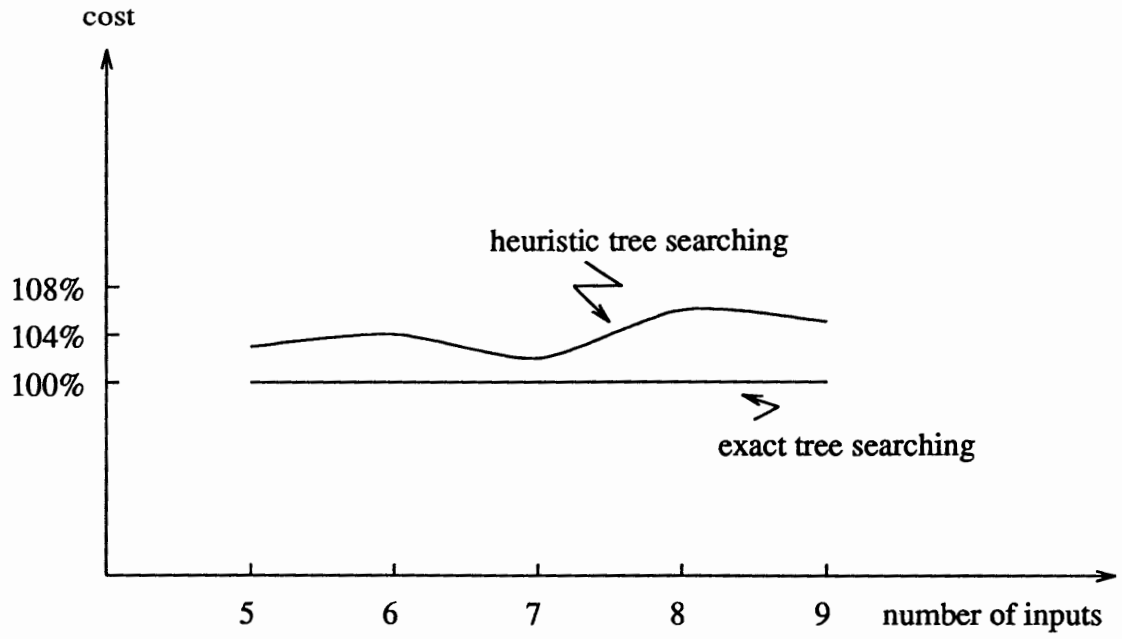


Figure 35. Cost of the heuristic searching versus the exact searching.

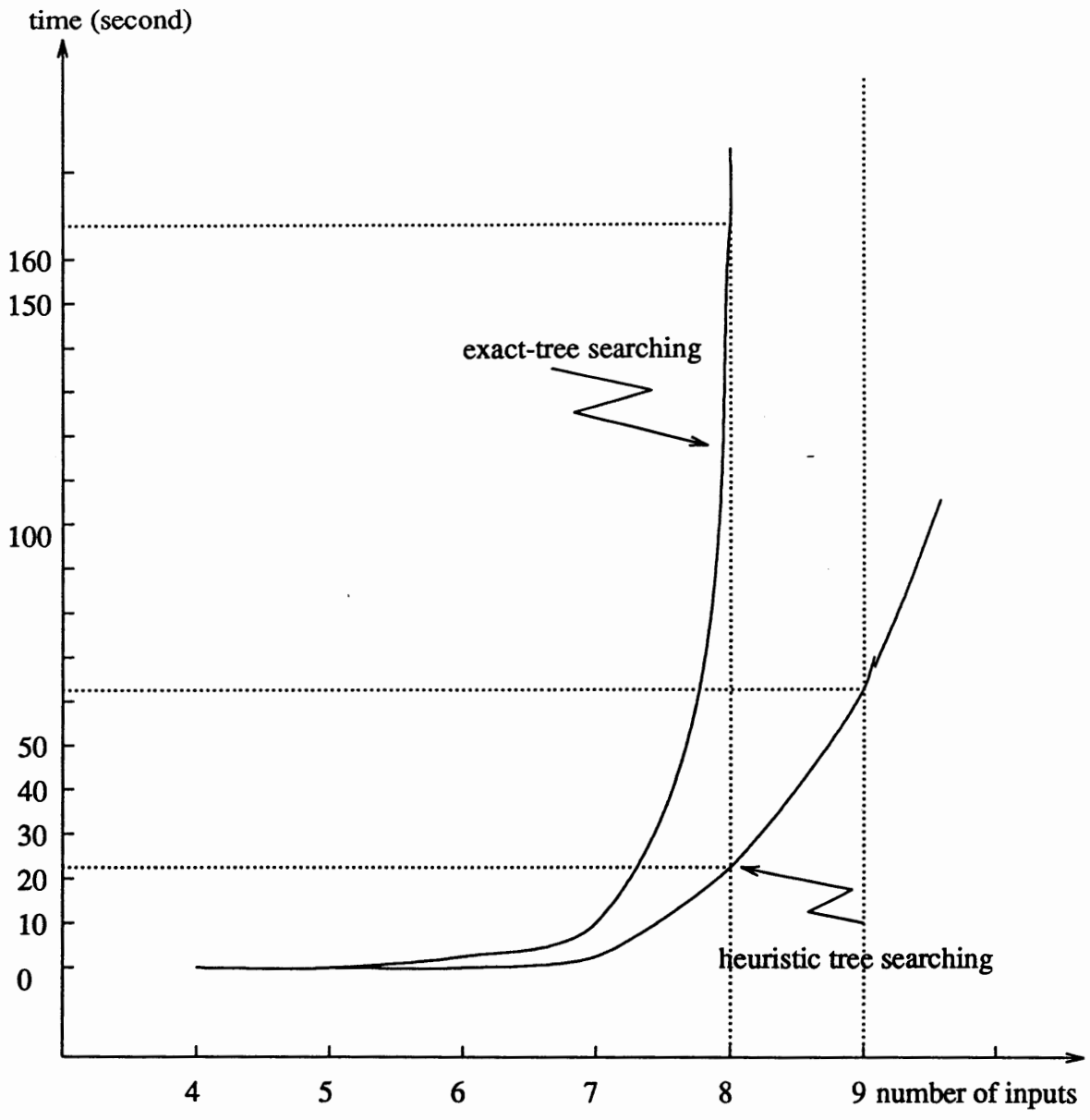


Figure 36. Real search time of exact searching versus heuristic searching.

TABLE II

GATE NUMBER COMPARISON OF EXACT SEARCHING AND RANDOM SEARCHING

Example Name	I/N	O/N	exact-tree searching		random order searching	
			# And-gate	# Exor-gate	# And-gate	#Exor-gate
misex25.tt	6	1	8	11	11	11
misex53.tt	6	1	6	7	10	7
misex57.tt	6	1	8	11	13	11
misex22.tt	6	1	8	11	15	11
misex55.tt	6	1	8	11	12	11
misex58.tt	6	1	6	7	10	7
misex56.tt	6	1	8	11	13	12
misex21.tt	6	1	10	15	15	16
5x6.tt	7	1	3	4	4	4
5x1.tt	7	1	13	15	26	15
5x5.tt	7	1	5	6	7	8
z44.tt	7	1	2	2	2	2
con11.tt	7	1	7	10	9	10
con12.tt	7	1	5	7	5	7
f55.tt	8	1	3	4	4	4
f54.tt	8	1	5	6	8	8
f53.tt	8	1	9	10	17	15
f52.tt	8	1	14	18	33	26
f51.tt	8	1	25	30	50	44

* I/N -- input number .

* O/N -- output number .

* #AND-gate -- the number of AND-gates .

* #EXOR-gate -- the number of EXOR-gates .

TABLE III

TIME COMPARISON OF EXACT SEARCHING AND HEURISTIC SEARCHING

Example Name	I/N	O/N	exact-tree searching				heuristic searching			
			# And	# Exor	time	level	# And	#Exor	time	level
misex25.tt	6	1	8	11	2.1	6	9	11	0.4	6
misex53.tt	6	1	6	7	1.4	5	8	7	0.3	5
misex57.tt	6	1	8	11	1.9	6	9	11	0.4	6
misex22.tt	6	1	8	11	2.3	6	9	11	0.4	6
misex55.tt	6	1	8	11	2.3	6	10	11	0.5	6
misex58.tt	6	1	6	7	1.8	5	6	7	0.3	5
misex56.tt	6	1	8	11	2.5	6	10	11	0.5	6
misex21.tt	6	1	10	15	3.1	6	10	15	0.6	6
5x6.tt	7	1	3	4	3.0	4	3	4	0.4	4
5x1.tt	7	1	13	15	21.3	7	14	15	3.9	7
5x5.tt	7	1	5	6	7.3	4	5	6	1.3	4
z44.tt	7	1	2	2	1.3	2	2	2	0.1	2
con11.tt	7	1	7	10	8.2	6	8	10	1.2	6
con12.tt	7	1	5	7	7.7	4	5	7	1.4	4
f55.tt	8	1	3	4	24.3	4	4	4	0.2	4
f54.tt	8	1	5	6	45.4	5	5	6	6.1	5
f53.tt	8	1	9	10	102.0	6	10	11	15.4	6
f52.tt	8	1	14	18	243.2	7	16	19	32.6	7
f51.tt	8	1	25	30	426.3	8	26	31	57.1	8
9nsy.tt	9	1	41	73	13760.0	9	48	73	110.2	9
misex64.tt	10	1					147	295	321.6	10
misex47.tt	11	1					53	39	573.2	11
misex60.tt	12	1					33	15	265.0	12
f13.tt	13	1					40	23	9.6	8
f14.tt	14	1					63	27	25.5	12
f15.tt	15	1					117	131	103.7	11
f16.tt	16	1					94	95	23.7	10
f17.tt	17	1					392	257	193.2	11
f18.tt	18	1					224	199	53.6	12
f19.tt	19	1					347	287	6.6	13
f20.tt	20	1					762	575	185.7	15

* I/N -- input number .
* O/N -- output number .
* #AND -- the number of AND-gates .
* #EXOR -- the number of EXOR-gates .

TABLE IV

GATE NUMBER COMPARISON OF EXACT PRM SEARCHING AND RM SEARCHING

Example Name	I/N	O/N	exact-tree searching				RM-tree searching			
			# And	# Exor	time	level	# And	#Exor	time	level
misex25.tt	6	1	8	11	2.1	6	10	10	0.1	6
misex53.tt	6	1	6	7	1.4	5	10	7	0.0	5
misex57.tt	6	1	8	11	1.9	6	12	9	0.0	6
misex22.tt	6	1	8	11	2.3	6	18	11	0.0	6
misex55.tt	6	1	8	11	2.3	6	20	11	0.1	6
misex58.tt	6	1	6	7	1.8	5	10	7	0.0	5
misex56.tt	6	1	8	11	2.5	6	14	11	0.0	6
misex21.tt	6	1	10	15	3.1	6	15	15	0.0	6
5xl.tt	7	1	13	15	21.3	7	25	14	0.1	7
5x5.tt	7	1	5	6	7.3	4	10	6	0.0	4
z44.tt	7	1	2	2	1.3	2	3	4	0.0	3
con11.tt	7	1	7	10	8.2	6	8	10	0.0	6
con12.tt	7	1	5	7	7.7	4	7	7	0.0	4
f52.tt	8	1	14	18	243.2	7	28	24	0.0	7
f51.tt	8	1	25	30	426.3	8	40	35	0.2	8

* I/N -- input number .

* O/N -- output number .

* #AND -- the number of AND-gates .

* #EXOR -- the number of EXOR-gates .

* time -- CPU time .

CHAPTER VII

CONCLUSIONS

A practical program with exact and quasi-minimum options for the minimization of PRM Trees, RM Trees to be realized in cellular-logic FPGAs has been presented. For realizing a given Boolean function, the circuit obtained from the exact PRM Tree has the minimum number of gates. When dealing with logic functions having many input variables, the heuristic PRM Tree searching works idealy to decrease the time complexity as well as provide quasi-minimum PRM circuit. Also, after some polarity transform of the original function, the program could be able to find the GRM Tree circuits. This gives us more opportunities to use the program to generate some other kind of tree-like circuits. The main advantages of our approach is that the presented program generates regular multi-level circuits based on the AND/EXOR gate structure, thus, the circuits can be easily placed to the CLI6000 chips. Moreover, the RM circuits obtained from the RM Trees are better suited the bussing network of the CLI6000 architecture, so, they can be easier routed after the placement.

The synthesis concept presented in this thesis open a wide area of interesting and new applications, especially to new FPGAs.

REFERENCES

1. A. Bedarida, S. Ercolani, and G. DeMicheli, "A New Technology Mapping Algorithm for the Design and Evaluation of Fuse/Antifuse-based Field-Programmable Gate Arrays," *1st ACM Workshop on FPGAs*, pp. 103-108, Berkeley, CA, February 1992.

2. R. J. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs," *28th ACM/IEEE Design Automation Conference*, pp. 227-233, San Francisco, CA, June 1991.

3. R. J. Francis, J. Rose, and Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," *ICCAD-91*, pp. 568-571, Santa Clara, CA, November 1991.

4. M. Fujita and Y. Matsunaga, "Multi-level Logic Minimization based on Minimal Support and its Application to the Minimization of Look-up Table Type FPGAs," *ICCAD-91*, pp. 560-563, Santa Clara, CA, November 1991.

5. R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "An Improved Synthesis Algorithm for Multiplexor-based PGAs," *1st ACM Workshop on FPGAs*, pp. 97-102, Berkeley, CA, February 1992.

6. P. Sawkar and D. Thomas, "Technology Mapping for Table-Look-Up based Field Programmable Gate Arrays," *1st ACM Workshop on FPGAs*, pp. 83-88, Berkeley, CA, February 1992.

7. R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proc of the IEEE.*, vol. 78, no. 2, pp. 264-300, February 1990.

8. Concurrent Logic Inc., "CLi6000 Series Field Programmable Gate Array," *Preliminary Information*, December 1991.

9. Algotronix Ltd., *Configurable Array Logic User Manual*, Edinburgh, UK, 1991.

10. Texas Instruments, *TPC10 Series 1.2-um Field-Programmable Gate Arrays*, July 1991.

11. S. M. Reddy, "Easily Testable Realizations for Logic Functions," *IEEE Trans. on Comput.*, vol. 21, no. 11, pp. 1183-1188, November 1972.

12. Lifei Wu and M. Perkowski, "Minimization of Permuted Reed-Muller Tree for Cellular Logic Programmable Gate Arrays," *2nd Int. Workshop on FPGAs*, Vienna, Austria, September 1992.

13. R. L. Ashenhurst, "The Decomposition of Switching Functions," *Proc. Int'l Symp. Theory of Switching Functions*, pp. 74-116, 1959.

14. H. A. Curtis, "Generalized Tree Circuit - The Basic Building Block of an Extended Decomposition Theory," *Journal of ACM*, vol. 10, pp. 562-581, 1963.

15. M. Davio, J. P. Deschamps, and A. Thayse, in *Discrete and Switching Functions*, McGraw-Hill, 1978.

16. M. A. Perkowski and P. D. Johnson, "Canonical Multi-Valued Input Reed-Muller Trees and Forms," *3rd NASA Symposium on VLSI Design*, 1991.

17. D. H. Green, "Reed-Muller Canonical Forms With Mixed Polarity and Their Manipulations," *Proc. of IEE Pt. E*, vol. 137, no. 1, January 1990.

18. D. H. Green, "Families of Reed-Muller Canonical Forms," *Int. J. of Electronics*, vol. 63, no. 2, pp. 259-280, January 1991.

19. M. A. Perkowski, L. Csanky, A. Sarabi, and I. Schäfer, "Minimization of Mixed-Polarity Canonical AND/EXOR Forms," *Proc. ICCD'92*, October, 1992.